

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

『赢在京东』系列技术教程

Presto 技术内幕

JD-Presto 研发团队 著

掌握大数据核心技术，
引领大数据实时计算潮流，京东技术大揭秘



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



翁志

现任京东首席技术顾问,信息安全部负责人,硅谷研发中心负责人,主要负责京东的技术创新,信息安全以及对外技术交流合作等,兼任全国信息安全标准化技术委员会委员。拥有近20年IT技术开发经验,拥有十余项国内外专利。2013年加入京东,领导京东大数据开源计算框架Presto荣获“2015COPU开源优秀项目奖”,本人获“中国开源杰出人物”称号。



机构负责人-戴东东

时任京东数据库管理部负责人,负责数据库、大数据等新技术研究工作,拥有近15年IT技术研发及团队管理经验,侧重于海量数据库、大数据、机器学习等技术领域的研究。2012年加入京东,组建并带领JD-Presto研发团队荣获多个国内外奖项。



高级架构师-吕信

Presto中国社区创建者与管理者,主要从事Hadoop、JD-Presto大数据开源产品及京东数据云的研发与架构设计,具有丰富的大数据解决方案及大数据公有云解决方案的设计及开发经验。



Presto 技术内幕

JD-Presto 研发团队 著



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

Presto是专门为大数据实时查询计算而设计和开发的产品。由于Presto是基于Java语言开发的,因此,对使用者和开发者而言,Presto极易学习、使用并针对特定的业务场景进行改造开发和性能优化。无论是对多数据源支持,还是高性能、易用性、可扩展性等方面,Presto都是大数据实时查询计算产品中的佼佼者。

本书按照由浅入深的顺序对Presto进行了全方位的细致讲解,具体内容包括Presto概述、Presto安装与部署、Presto RESTful框架解析、提交查询、生成查询执行计划、查询调度、查询执行、队列、System Connector、Hive Connector、Kafka Connector、Connector开发、Functions开发、JD-Presto功能改造、Presto性能调优、Presto应用场景。

本书对Java研发工程师,在大数据技术领域从业的研发工程师、运维工程师、架构师,对大数据研究方向感兴趣的大学老师和学生,以及大数据技术的入门学者,从事海量数据分析应用的开发、研究者都有很强的参考价值。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Presto 技术内幕 / JD-Presto 研发团队著. —北京: 电子工业出版社, 2016.7

“赢在京东”系列技术教程

ISBN 978-7-121-28563-9

I. ①P… II. ①J… III. ①数据处理—教材 IV. ①TP274

中国版本图书馆 CIP 数据核字(2016)第 073676 号

策划编辑: 张慧敏

责任编辑: 王 静

印 刷: 三河市兴达印务有限公司

装 订: 三河市兴达印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×1092 1/16 印张: 19.5 字数: 490 千字

版 次: 2016 年 7 月第 1 版

印 次: 2016 年 7 月第 1 次印刷

定 价: 69.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: 010-51260888-819 faq@phei.com.cn。

序言 1

12 年来，京东一直以开源技术作为构建自身核心技术体系的基础，在开源领域深耕多年，积极推动行业分享与交流。JD-Presto 是京东首个贡献于开源社区的软件产品，开创了京东回馈开源社区的先河。

一年前，JD-Presto 研发团队是京东众多研发团队中的普通一员，时至今日，他们已成长为京东负有盛名的研发团队之一。一年来，他们忍受着孤独与寂寞，面对着无数困难与挫折，但他们从未放弃，凭着对技术的热爱、对信念的坚守、对优秀产品的执着追求，勇敢面对困难与挑战，历尽艰辛，终于迎来胜利的曙光。目前 JD-Presto 在京东内部已成功应用于精准营销、安全识别、商家后台、财务报表等十多个生产系统。京东为拥有 JD-Presto 研发团队而骄傲！

与书店里的 Hadoop 书籍不同的是，本书是市面上第一本详细介绍 Presto 的书籍。它由浅入深地详细介绍了 Presto 的安装过程、内部运行原理机制、功能特性、性能优化方法，以及在应用过程中常见的问题及解决方案等，同时详细介绍了 JD-Presto 版本的许多新功能及其适用的业务场景。这些功能的成功研发，使得 JD-Presto 版本不仅可用于离线大数据分析计算来提升性能，还可用于多种数据源混合进行实时大数据分析计算，使一些使用其他大数据技术不能解决的业务场景有了一个全新有效的解决方案。同时，他们维护着 Presto 在中国社区的建设，鼓舞着和培养了一群 Presto 的爱好者。

今天，国内已有越来越多的技术人员与公司在使用的 JD-Presto 版本，相信本书能对读者学习和使用 JD-Presto 版本提供很大的帮助。本书不仅仅满足了初学者对技术书籍的渴求，也适合对 JD-Presto 版本有一定使用经验的人员学习参考使用。希望它能成为你在 Presto 技术旅程中的一个朋友，点亮你前进的道路。

张晨

京东集团 CTO

序言 2

记得我刚加入京东的时候，Hadoop 的集群建设方兴未艾，大数据计算纷纷被转到 Hadoop 中进行处理。然而，以前做数据挖掘的同学并不十分精通 Java 语言。他们更擅长于业务和 SQL 语言。为了让他们的工作更有效率、更得心应手，有必要找寻一个像 Google Dremel 一样，以 SQL 为驱动的交互式 Ad-hoc 查询工具。于是我花了一些时间进行评估，包括 Hive、Spark、Impala 等，后来 Facebook 刚开源不久的 Presto 便映入眼帘。Presto 设计简单精巧，可以处理海量数据，最大化地利用硬件性能，计算全部在内存中完成，很好地利用高速网络来进行数据调度，其编码风格一致，易于掌握和改进。虽然在当时 Presto 的数据源和 SQL 的支持度还有待丰富，但已经初步展示了发展的潜力，很适合京东的应用。Presto 就这样在京东找到了家。JD-Presto 研发团队具有多年 JAVA 编程和分布式并行处理经验，他们工作兢兢业业，在原先开源的基础上，丰富了多种数据源的处理接口，并且通过对任务调度的追踪和分析，改进并提升了并行处理任务的吞吐量。我们积极和 Facebook 的 Presto 团队合作，成为 Presto 在国内的首批贡献者，建立并运行 Presto 的中国社区。我们也和 CSDN 的同仁建立了 Presto 的开发者论坛，在国内积极推广，普及 Presto 知识，让更多的开发人员受益。

京东是开源社区的受益者，也是开源开放文化的拥护者。我们将自己取得的点点滴滴毫无保留地回馈于开源社区，与大家共享，和大家一起进步。为了更好地服务于对 Presto 有兴趣的研发人员，京东的同事们将平时工作中的积累和认识编辑成书，奉献给大家，希望他们的努力有助于读者学习和掌握 Presto。在此，我特别感谢为此书出版而辛勤工作的同事，他们是吕信、郭李明、袁安峰、孔云龙、戴东东等人。

翁志

京东集团首席技术顾问

序言 3

Presto is an open source, highly scalable, distributed SQL engine we developed at Facebook. Since releasing it two years ago, we have witnessed a tremendous amount of community interest and adoption. In the last twelve months, the Presto community has grown from technology firms in Silicon Valley to a broad variety of organizations worldwide. Today, NASDAQ, the second largest stock exchange in the United States, runs Presto to power their analytical use cases.

（Presto 是我们在 Facebook 开发的开源、高度可扩展的分布式 SQL 引擎。Presto 在两年前发布后，大量的社区开始关注和后续应用 Presto。在最近的 12 个月里，Presto 社区已经从一家硅谷科技公司成长为全球性的生态圈，涵盖一批多样化的组织。如今，全美第二大证券交易所纳斯达克也在使用 Presto 来驱动他们的案例分析业务。）

As organizations adopt Presto, many are contributing back to it. Teradata, a leader in data warehousing technology, maintains a group of engineers who contribute to Presto. We have also collaborated with engineers at JD.com since 2014 to make Presto better. Presto earned an "Open Source Outstanding Project" award at the 2015 China Open Source World Summit.

（组织机构们不仅在应用 Presto，他们中的一部分也在回馈 Presto。Teradata，数据仓库技术领域内的引领者之一，拥有一支专门为 Presto 贡献力量的工程师团队。从 2014 年起，我们与中国电商领域引领者京东公司的开发人员展开合作，精益求精地打造 Presto 产品。随后 Presto 在 2015 年“第十届开源中国开源世界高峰论坛”上获取“2015 COPU 开源优秀项目大奖”。）

We hope you will learn more about Presto through this book developed by engineers at JD.com, and we encourage you to work with the community to improve Presto.

（我们真诚地希望你通过这本由京东工程师们撰写的书更深入地了解 Presto 引擎，也鼓励并期待你携手社区，改善 Presto 引擎。）

Jay Tang

Manager of Presto product, Big Data, Facebook

前言

为什么要写这本书

相信很多软件开发人员在遇到应用系统涉及需要多种数据源、多种字符集下的混合计算时，都被数据的准确性和查询性能所困扰，一直以来都没有一个有效的技术方案能解决这个问题。2014年7月，京东CTO体系首席技术顾问翁志先生把Presto引入京东，数据准确性验证表明，Presto在多种数据源、多种字符集下的混合计算均表现优异，数据准确性符合财务标准。我们认识到，Presto是一个非常优秀的产品，尽管它有许多不足，我们决心把它打造为一个介于T+1计算和生产实时报表之间混合计算的大数据产品，支持秒级、分钟级获取查询结果、支持多并发的交互式Ad-Hoc查询与分析。Presto是一个较为完美的大数据实时计算解决方案，而市面上这类产品太少了，我们将经过生产环境验证的JD-Presto版本开源给业内使用，主导Presto中国社区的推广工作。应广大Presto技术爱好者的要求，我们编写了这本书，以帮助更多的初学者、大数据领域的从业者学习及使用JD-Presto。

和其他许多调研过Presto的部门或公司一样，我们最初搭建的Presto测试集群在京东这种大数据量的环境下复杂计算任务几乎运行不起来，JOIN查询也比较弱，集群时常宕机。同时，在测试过程中我们也发现，与Spark、Impala相比，Presto使用更为简单、易用、高效，这让我们深感振奋，一致认为这个产品值得团队花大力气去研究和优化。因此，我们专门申请了上百台高性能服务器来搭建性能测试集群环境，随后在几百TB级数据量下进行大规模的测试与验证，完全证实了Presto的平均性能是Hive等离线数据分析产品性能的10倍，并且支持ANSI标准SQL、数据源完全解耦等一系列优势。在之后一年多的时间中，我们研发了众多实用功能，着力修复了许多Bug，大幅提升了Presto的查询性能和并发执行性能，如重写了数据库驱动，使得Presto连接数据库进行海量数据查询的性能提升了几百倍；优化了内存使用效率，使查询性能对内存大小的依赖大幅降低；支持分布式数据库的分库分表设计；优化数据存储算法。通过上述一系列新功能的成功研发和对性能的大幅改进，使得JD-Presto的性能稳定是Hive的10倍以上，是Spark的3倍以上，集群成本节省一半，在一个大规模的大数据集群中，这种技术优势可带来显著的经济效益。

时至今日，京东已有近二十多个系统在使用 JD-Presto 版本，尤其是在精准营销平台中 JD-Presto 作为大数据即席运算平台，起到了关键性作用，极大提升了采销部门进行精准营销活动的效果和效率。在团队的努力下，有越来越多的公司使用了我们的产品，取得良好的社会效益并扩大了京东的技术影响力。

JD-Presto 研发团队先后荣获多个奖项：在“第十届开源中国开源世界高峰论坛”上荣获“2015 COPU 开源优秀项目奖”；在“第十四届中日韩三国 IT 局长 OSS 会议及东北亚开源软件推进论坛”上荣获“优秀项目技术奖”；在京东第四届 CTO 体系“金项奖”中荣获第一名。JD-Presto 产品蕴含着团队中这群不羁的兄弟们的心血、激情和梦想，从做产品研发决定的那一刻起，就注定我们不平凡。

本书适合的读者对象

- Java 研发工程师；
- 在大数据技术领域从业的研发工程师、运维工程师、架构师；
- 对大数据研究方向感兴趣的大学老师和学生，以及大数据技术的入门学者；
- 从事海量数据分析应用的开发者、研究者。

如何阅读本书

本书按照由浅入深的顺序对大数据实时计算产品 Presto 进行了全方位的细致讲解，从基本的安装部署与使用到源码解析，以及根据实际业务需求进行新功能改造开发和性能优化。

若你从来没有使用过 Presto，那么完全可以按照由浅入深的顺序从头开始阅读此书。

若你已经成功地使用了 Presto 一段时间，想要深入地了解 Presto 的设计思想并希望从源码级别了解 Presto 的实现思路，那么可以直接从核心设计篇进行阅读。

若你本来就是 Presto 的贡献者，苦于 Presto 不能满足你的实际业务问题，想要对 Presto 进行功能改造但是还没有明确的思路；又或者你在使用 Presto 的过程中发现其性能没有达到你的预期，而又找不到优化的方案，那么高级篇会给你提供一些建议。

在本书最后的附录部分不仅对 Presto 使用过程中出现的各种问题给出了明确的解决方案，还对 Presto 中的各个配置参数的含义和作用进行了详细的说明并给出了推荐配置值。

相信此书会为广大大数据爱好者提供一个新的视角，思考并解决大数据相关问题。

勘误和支持

由于作者水平和能力有限，编写时间仓促，本书存在不妥之处在所难免，我们真诚希望同行和广大读者朋友们不吝赐教。另外，如果你有关于 JD-Presto 的任何问题，可以加入我们的 QQ 群：141254058，与我们在线交流，或者访问网站：<http://prestodb-china.com/>，期待你的反馈意见，我们将不胜感激。

致谢

感谢现任京东 CTO 体系首席技术顾问、信息安全部、京东硅谷研发中心负责人翁志先生，在产品研发和推广使用的过程中给予了我们极大的支持和鼓励。同时在本书出版的过程中，从选题、审稿到出版无不得到他的热心帮助，在此致以深深的谢意！

感谢京东云平台——数据研发运营部的负责人樊建刚先生给予我们充分的授权和支持，他鼓励我们将产品开源、组建技术交流群和中文社区，使得 JD-Presto 成为京东第一个开源的软件产品，开创京东回馈开源社区的先河。

感谢研发团队的兄弟们，他们酷爱技术、注重团队协作、深入理解客户需求，能从一个较为全面的角度考虑产品的架构设计与功能研发，这是产品成功的基石，也是最关键的因素。

回首这一路的艰辛，无以表达此时的心情，只能说，兄弟们所做的一切都让我心怀崇敬！

戴东东
2016 年 4 月

目 录

第一部分 基础篇

第 1 章 Presto 概述.....	2
1.1 Presto 背景及发展.....	2
1.2 Presto 特点.....	2
1.3 基本概念.....	3
1.3.1 Presto 服务进程.....	3
1.3.2 Presto 模型.....	4
1.3.3 Presto 查询执行模型.....	5
1.4 Presto 整体架构.....	9
1.4.1 硬件架构.....	9
1.4.2 软件架构.....	9
1.5 小结.....	11
第 2 章 Presto 安装与部署.....	12
2.1 环境说明.....	12
2.2 准备工作.....	13
2.2.1 建立 SSH 信任关系.....	13
2.2.2 安装 Java.....	17
2.2.3 安装 Maven.....	17
2.2.4 安装 Hive.....	18
2.3 源码编译.....	21
2.3.1 下载源码.....	21
2.3.2 源码结构说明.....	22

- 2.3.3 编译 26
- 2.4 部署 30
 - 2.4.1 服务部署 30
 - 2.4.2 客户端部署 38
 - 2.4.3 JDBC 使用 40
- 2.5 小结 42

第二部分 核心设计篇

- 第 3 章 Presto RESTful 框架解析 44
 - 3.1 Statement 服务接口 44
 - 3.2 Query 服务接口 47
 - 3.3 Stage 服务接口 48
 - 3.4 Task 服务接口 49
 - 3.5 小结 52
- 第 4 章 提交查询 53
 - 4.1 提交查询的步骤 53
 - 4.2 源码解析 53
 - 4.3 提交查询的流程 60
 - 4.4 小结 61
- 第 5 章 生成查询执行计划 62
 - 5.1 基本概念 63
 - 5.1.1 Node 63
 - 5.1.2 Metadata API 67
 - 5.2 词法与语法分析 68
 - 5.2.1 语法规则 69
 - 5.2.2 词法分析 69
 - 5.2.3 语法分析 71
 - 5.3 获取 QueryExecution 72
 - 5.3.1 获取 QueryExecutionFactory 72
 - 5.3.2 创建 QueryExecution 73
 - 5.3.3 启动 QueryExecution 74
 - 5.4 语义分析 77
 - 5.4.1 Statement 分析 77
 - 5.4.2 Relation 分析 84
 - 5.4.3 表达式分析 91

5.5	执行计划生成.....	91
5.5.1	执行计划节点	91
5.5.2	SQL 执行计划	93
5.5.3	Relation 执行计划	95
5.5.4	Query 执行计划	99
5.6	执行计划优化.....	102
5.6.1	ImplementSampleAsFilter	102
5.6.2	CanonicalizeExpressions	102
5.6.3	SimplifyExpressions	102
5.6.4	UnaliasSymbolReferences	103
5.6.5	PruneRedundantProjections	103
5.6.6	SetFlatteningOptimizer	103
5.6.7	LimitPushDown	104
5.6.8	PredicatePushDown	104
5.6.9	MergeProjections	104
5.6.10	ProjectionPushDown	104
5.6.11	IndexJoinOptimizer	105
5.6.12	CountConstantOptimizer	105
5.6.13	WindowFilterPushDown	105
5.6.14	HashGenerationOptimizer	105
5.6.15	PruneUnreferencedOutputs	106
5.6.16	MetadataQueryOptimizer	106
5.6.17	SingleDistinctOptimizer	106
5.6.18	BeginTableWrite	106
5.6.19	AddExchanges	107
5.6.20	PickLayout	107
5.7	执行计划分段.....	107
5.7.1	Source	107
5.7.2	Fixed	107
5.7.3	Single	107
5.7.4	Coordinator_only	107
5.8	示例.....	108
5.8.1	Count 执行计划	108
5.8.2	Join 执行计划	108
5.9	小结.....	110

- 第 6 章 查询调度111
 - 6.1 生成调度执行器 111
 - 6.2 查询调度过程 113
 - 6.2.1 NodeManager 114
 - 6.2.2 NodeSelector 115
 - 6.3 小结 118
- 第 7 章 查询执行 119
 - 7.1 查询执行逻辑 120
 - 7.2 Task 调度 120
 - 7.2.1 Source Task 调度 120
 - 7.2.2 Fixed Task 调度 126
 - 7.2.3 Single Task 调度 128
 - 7.2.4 Coordinator_Only Task 调度 128
 - 7.3 Task 执行 129
 - 7.3.1 创建 Task 129
 - 7.3.2 更新 Task 135
 - 7.3.3 运行 Task 140
 - 7.4 小结 147
- 第 8 章 队列 148
 - 8.1 配置说明 148
 - 8.1.1 queues 队列定义 149
 - 8.1.2 rules 规则定义 149
 - 8.2 队列加载 150
 - 8.3 队列匹配 151
 - 8.4 小结 154
- 第 9 章 System Connector 155
 - 9.1 System Connector 使用 155
 - 9.1.1 Information_schema 155
 - 9.1.2 Metadata 157
 - 9.1.3 Runtime 157
 - 9.2 System Connector 实现 159
 - 9.2.1 Information_schema 实现 160
 - 9.2.2 System Connector 实现 163
 - 9.3 小结 168

第 10 章	Hive Connector	169
10.1	与 Hive 的结合	170
10.2	Split 分片管理	175
10.3	数据读取	179
10.4	Create Table As Select 的实现	182
10.5	小结	186
第 11 章	Kafka Connector	187
11.1	认识 Kafka Connector	187
11.1.1	配置	187
11.1.2	配置属性	187
11.1.3	内置字段	189
11.1.4	表定义文件	190
11.1.5	Kafka 中的 key 和 message	191
11.1.6	行解码	192
11.1.7	日期和时间解码器	194
11.1.8	文本解码器	194
11.1.9	数值解码器	194
11.2	Kafka 连接器使用教程	194
11.2.1	安装 Apache Kafka	195
11.2.2	下载数据	195
11.2.3	在 Presto 中配置 Kafka topics	197
11.2.4	基本数据查询	197
11.2.5	添加表定义文件	199
11.2.6	将 message 中所有值映射到不同列	200
11.2.7	使用实时数据	202
11.3	Kafka Connector 获取数据	207
11.3.1	Split 分片管理	207
11.3.2	数据读取	209
11.4	小结	210
第 12 章	Connector 开发	211
12.1	创建 Maven 工程	211
12.2	注册 Plugin	213
12.3	Connector	213
12.4	Metadata	215
12.5	SplitManager	217

12.6	RecordSetProvider	218
12.7	小结	219
第 13 章	Functions 开发	220
13.1	Function 注册	220
13.2	窗口函数	225
13.3	聚合函数	229
13.4	小结	232
第三部分 高级篇		
第 14 章	JD-Presto 功能改造	234
14.1	PDBO 功能开发	234
14.1.1	JDBC Split 剖析	235
14.1.2	JdbcRecordCursor 剖析	238
14.1.3	分批次读取实现原理	240
14.1.4	动态步长实现原理	243
14.1.5	条件下发	245
14.1.6	PDBO 配置定义	247
14.2	DDL 及 DML 支持	250
14.2.1	Hive 连接器 Insert 功能	250
14.2.2	Hive 连接器 CTAS 动态分区表功能	252
14.3	动态增加、修改、删除 Catalog	254
14.3.1	目的	254
14.3.2	现状	254
14.3.3	实现	255
14.3.4	效果	258
14.4	小结	258
第 15 章	Presto 性能调优	259
15.1	合理设计分区	259
15.2	Group By 字句优化	259
15.3	使用模糊聚合函数	259
15.4	合并多条 Like 子句为一条 regexp_like 子句	260
15.5	大表放在 Join 子句左边	260
15.6	关闭 distributed hash join	261
15.7	使用 ORC 存储	261
15.8	小结	262

第 16 章 Presto 应用场景	263
16.1 ETL	263
16.2 实时数据计算	264
16.3 Ad-Hoc 查询	266
16.4 实时数据流分析	266
16.5 小结	268
附录 A 常见问题及解决办法	269
A.1 同时访问两个 Hadoop 集群	269
A.2 Kafka 集群重启后无法获取数据	272
A.3 Task exceeded max memory size	277
A.4 SQL 中 In 子句太长导致栈溢出错误	278
A.5 高并发导致大量查询出错	279
附录 B Presto 配置参数说明	282
附录 C Presto 执行信息说明	289

写在前面

IT 领域中的某些概念可能真的很难用中文表述清楚其英文的含义，因此为了保证读者能够准确理解某些关键术语和概念的含义，在本书中对于一些关键的概念或者名词就不再翻译为对应的中文了。

由于本书中某些章节对源码进行了解析说明，本书中的所有源码都是针对于 Presto-0.107 进行讲解的，并且在对源码或者原理讲解时，仅写出了关键源码，对于非关键源码，没有进行书写。

第一部分 基础篇

第 1 章

Presto 概述

1.1 Presto 背景及发展

随着经济和科技的快速发展，各大企业和公司需要处理的数据在飞速增长，因此，大数据技术在最近几年突然兴起并且迅速变得火热。在大数据技术的浪潮中，出现了很多优秀的大数据存储和计算产品，其中最具影响力的就是 Hadoop。Hadoop 提供了大数据存储和计算的一整套解决方案，完美地解决了大数据的存储和计算问题。但是 Hadoop 提供的大数据解决方案使用的是 Map-Reduce 计算框架，这种计算框架适用于大数据的离线和批量计算，因为该计算框架强调的是吞吐率而不是计算效率，所以其不能满足大数据快速实时 Ad-Hoc 查询计算的性能要求。

各大互联网公司和企业也发现了 Map-Reduce 在进行快速实时 Ad-Hoc 查询计算时所表现出来的弊端，因此，开源社区和各大互联网公司纷纷进行大数据实时 Ad-Hoc 查询计算产品的研发，Facebook 于 2012 年秋季开始开发 Presto，目前该产品已经在超过 1000 名 Facebook 雇员中使用，每天运行超过 30000 个查询，每日查询数据量在 1PB 级别。Facebook 称 Presto 的性能比 Hive 要好上 10 倍还多，2013 年 Facebook 正式宣布开源 Presto。

1.2 Presto 特点

Presto 是专门为大数据实时查询计算而设计和开发的产品。由于 Presto 是基于 Java 语言开发的，因此，对使用者和开发者而言，Presto 极易学习、使用并针对于特定的业务场景进行改造开发和性能优化。无论是对多数据源支持，还是高性能、易用性、可扩展性等方面，Presto 都是大数据实时查询计算产品中的佼佼者，Presto 的特点如表 1-1 所示。

表 1-1

特 点	说 明
多数据源	目前 Presto 可以支持 My SQL、PostgreSQL、Cassandra、Hive、Kafka、JMX 等多种 Connector，除此之外，京东商城改造之后的 Presto 也能够很好地支持 Oracle、SQLServer，并且可以支持分库分表以及数据快速读取的功能
支持 SQL	Presto 已经可以完全支持 ANSI SQL，并提供了一个 SQL Shell 给用户，用户可以直接使用 ANSI SQL 进行数据查询和计算
扩展性	Presto 有很好的扩展性，开发人员可以很容易地开发出适用于自己特定数据源的 Connector，并且可以使用 SQL 语句查询和分析自定义 Connector 中的数据，读者可以参照第 12 章开发出自定义 Connector
混合计算	在数据库中每种类型的数据源都对应于一种特定类型的 Connector，用户可以根据业务需要在 Presto 中针对于一种类型的 Connector 配置一个或者多个 Catalog 并查询其中的数据，用户可以混合多个 Catalog 进行 join 查询和计算
高性能	经过 Facebook 和京东商城的测试，Presto 的查询的平均性能是 Hive 的 10 倍以上
流水线	由于 Presto 是基于 PipeLine 进行设计的，因此在进行海量数据处理的过程中，终端用户不用等到所有的数据都处理完毕才能看到结果，而是可以像自来水管一样，一旦计算开始，就可以立即产生一部分结果数据，并且结果数据会一部分接一部分地呈现在终端客户面前

1.3 基本概念

1.3.1 Presto 服务进程

Presto 集群中一共有两种服务器进程：Coordinator 服务进程和 Worker 服务进程，其中 Coordinator 服务进程的主要作用是：接收查询请求、解析查询语句、生成查询执行计划、任务调度和 Worker 管理。而 Worker 服务进程则执行被分解后的查询执行任务：Task。

1. Coordinator

Coordinator 服务进程部署于集群中一个单独的节点上，是整个 Presto 集群的管理节点。Coordinator 服务进程主要用于接收客户端提交的查询，查询语句解析，生成查询执行计划、Stage 和 Task 并对生成的 Task 进行调度。除此之外，Coordinator 还对集群中的所有 Worker 进行管理。Coordinator 进程是整个 Presto 集群的 Master 进程，该进程既与 Worker 进行通信从而获得最新的 Worker 信息，又与 Client 进行通信，从而接受查询请求，而所有的这些工作都是通过 Coordinator 上的 StatementResource 类提供的 RESTful 服务来完成的。

2. Worker

在一个 Presto 集群中，存在一个 Coordinator 节点和多个 Worker 节点。Coordinator 节点是管理节点，而 Worker 节点就是工作节点。在每个 Worker 节点上都会存在一个 Worker 服务进程，该服务进程主要进行数据的处理以及 Task 的执行。Worker 服务进程每隔一定的时间都会向 Coordinator 上的 RESTful 服务发送心跳，从而告知 Coordinator：我还活着，并接

受你的调度。当客户端提交一个查询的时候，Coordinator 则会从当前存活的 Worker 列表中选择出合适的 Worker 节点去运行 Task。而 Worker 在执行每个 Task 的时候又会进一步对当前 Task 读入的每个 Spill 进行一系列的操作和处理。

1.3.2 Presto 模型

Presto 可以通过多种不同类型的 Connector 访问多种数据源，目前支持的 Connector 包括：Hive、JMX、MySQL、Cassandra、PostgreSQL 以及 Kafka。在本节会说明 Presto 是如何访问不同类型数据源的，并对 Presto 中的模型和概念进行描述。

1. Connector

Presto 是通过多种多样的 Connector 来访问多种不同的数据源的。你可以将 Connector 当作 Presto 访问各种不同数据源的驱动程序。一般情况下，Presto 针对每种数据源都有与之对应的 Connector。每种 Connector 都实现了 Presto 中标准的 SPI 接口，因此只要你实现 Presto 中标准的 SPI 接口，就可以轻易地实现使用适合自己特定需求的 Connector 来访问特定的数据源。Presto 目前可以支持的 Connector 有 Hive、Kafka、JMX、MySQL、Cassandra 等，都有其对应的 Build-In Connector（内置的 Connector）。

当你需要使用某种 Connector 访问特定的数据源时，需要在 `$PRESTO_HOME/etc/catalog/` 中创建一个配置文件：example.properties（文件名字无限制，但是其后缀名必须为 .properties），在该配置文件中必须要设置一个属性：connector.name，该属性是必须设置的。Presto 中的 Connector Manager 就是通过该配置属性来决定使用哪个 Connector 去访问相应的数据源的。例如，你现在需要访问一个 Hive 数据源，那么你在配置文件中就需要将属性 connector.name 设置为 Hive-cdh5 或者 Hive-cdh4，这样 Presto 就会使用内置的 Hive connector 去访问 Hive 数据仓库中相应的数据。

2. Catalog

Presto 中的 Catalog 类似于 MySQL 中的一个数据库实例。而 Schema 就类似于 MySQL 中的一个 Database。通过使用特定的 Connector 访问 Catalog 中指定的数据源，一个 Catalog 中可以包含多个 Schema。那么怎么定义一个 Catalog 呢？其实你不需要特意去指定 Catalog。正如 1.3.2 节所述，假设你想要访问 Hive 中的数据，则需要在 `$PRESTO_HOME/etc/catalog/` 中创建一个配置文件：example.properties。该配置文件中定义了诸如 Hive metastore 的 URI 等访问 Hive 中的数据所需要的所有配置项，并且配置文件的名字就是 Catalog 的名称：example。从这里可以看出 Presto 中配置文件的名字（不带 .properties）就是 Catalog 的名字。

当你访问 Catalog 中的某个表时，该表的全名总是以 Catalog 的名字开始。例如名字为 example.schema1.table1 的表，指的是表 table1 位于名为 schema1 的 schema 中，而 schema1 又位于名为 example 的 Catalog 中。

3. Schema

Presto 中的 Schema 就类似于 MySQL 中的 Database。一个 Catalog 名称和一个 Schema 名称唯一确定了可以查询的一系列表的集合。当通过 Presto 去查询 Hive 或者 MySQL 中的数据时，你会发现 Presto 中的 Schema 与 Hive 或者 MySQL 中的 Database 是相对应的。

4. Table

Presto 中的 Table 与传统数据库中的 Table 的含义是一样的。

1.3.3 Presto 查询执行模型

Presto 在执行 SQL 语句时，将这些 SQL 语句解析为相应的查询，并在分布式集群中执行这些查询。

1. Statement

Statement 语句。其实就是指我们输入的 SQL 语句。Presto 支持符合 ANSI 标准的 SQL 语句。这种语句由子句（Clause）、表达式（Expression）和断言（Predicate）组成。

Presto 为什么将语句（Statement）和查询（Query）的概念分开呢？

因为在 Presto 中，语句和查询本身就是不同的概念。语句指的是终端用户输入的用文字表示的 SQL 语句；当 Presto 执行输入的 SQL 语句时，会根据 SQL 语句生成查询执行计划，进而生成可以执行的查询（Query），而查询代表的是分布到所有的 Worker 之间执行的实际查询操作。

2. Query

Query 即查询执行。当 Presto 接收一个 SQL 语句并执行时，会解析该 SQL 语句，将其转变成一个查询执行和相关的查询执行计划。一个查询执行代表可以在 Presto 集群中运行的查询，是由运行在各个 Worker 上且各自之间相互关联的阶段（Stage）组成的。

那么 SQL 语句与查询执行之间有什么不同呢？

其实很简单，你可以认为 SQL 语句就是提交给 Presto 的用文字表示的 SQL 执行语句。而查询执行则是为了完成 SQL 语句所表述的查询而实例化的配置信息、组件、查询执行计划和优化信息等。一个查询执行由 Stage、Task、Driver、Split、Operator 和 DataSource 组成。这些组件之间通过内部联系共同组成了一个查询执行，从而得到 SQL 语句表述的查询，并得到相应的结果集。

3. Stage

Stage 即查询执行阶段。当 Presto 运行 Query 时，Presto 会将一个 Query 拆分成具有层级关系的多个 Stage，一个 Stage 就代表查询执行计划的一部分。例如，当我们执行一个查询，从 Hive 的一张具有 1 亿条记录的表中查询数据并进行聚合操作时，Presto 会创建一个 Root Stage（在后面的章节你会知道，该 Stage 就是 Single Stage），该 Stage 聚合其上游 Stage 的输出数据，然后将结果输出给 Coordinator，并由 Coordinator 将结果输出给终端用户。

通常情况下, Stage 之间是树状的层级结构。每个 Query 都有一个 Root Stage。该 Stage 用于聚集所有其他 Stage 的输出数据,并将最终的数据反馈给终端用户。需要注意的是,Stage 并不会在集群中实际执行,它只是 Coordinator 用于对查询执行计划进行管理和建模的逻辑概念。每个 Stage (除了 Single Stage 和 Source Stage) 都会有输入和输出,都会从上游 Stage 读取数据,然后将产生结果输出给下游 Stage。需要注意的是: Source Stage 没有上游 Stage, 它从 Connector 获取数据。Single Stage 没有下游 Stage, 它的结果直接输出给 Coordinator, 并由 Coordinator 输出给终端用户。

Presto 中的 Stage 共分为 4 种, 具体介绍如下。

Coordinator_Only: 这种类型的 Stage 用于执行 DDL 或者 DML 语句中最终的表结构创建或者更改。

Single: 这种类型的 Stage 用于聚合子 Stage 的输出数据, 并将最终数据输出给终端用户。

Fixed: 这种类型的 Stage 用于接受其子 Stage 产生的数据并在集群中对这些数据进行分布式的聚合或者分组计算。

Source: 这种类型的 Stage 用于直接连接数据源, 从数据源读取数据, 在读取数据的时候, 该阶段也会根据 Presto 对查询执行计划的优化完成相关的断言下发 (Predicate PushDown) 和条件过滤等。

说明

由于一个 SQL 查询可以被分解为多个前后关联的 Stage, 在这里我们约定: 按照数据的流向, 越靠近数据源的 Stage 越处于上游, 越远离数据源的 Stage 越处于下游。

4. Exchange

Exchange 的字面意思就是“交换”。Presto 的 Stage 是通过 Exchange 来连接另一个 Stage 的。Exchange 用于完成有上下游关系的 Stage 之间的数据交换。在 Presto 中有两种 Exchange: Output Buffer 和 Exchange Client。生产数据的 Stage 通过名为 Output Buffer 的 Exchange 将数据传送给其下游的 Stage (根据数据的流向, 分为上下游, 你可以将 Presto 中查询执行过程中的数据比喻成一条河流, 那么产生数据的 Stage 相对于消费数据的 Stage 来说, 就是上游)。消费数据的 Stage 通过名为 Exchange Client 的 Exchange 从上游 Stage 读取数据。

如果当前的 Stage 是 Source 类型的 Stage, 那么该 Stage 则是直接通过相应的 Connector 从数据源读取数据的。而该 Stage 则是通过名为 Source Operator 的 Operator 与 Connector 进行交互的。例如, 如果一个 Source Stage 直接从 HDFS 获取数据, 那么这种操作不是通过 Exchange Client 来完成的, 而是通过运行于 Driver 中的 Source Operator 来完成的。

5. Task

从前面的章节我们可以知道, Stage 并不会在 Presto 集群中实际运行, 它仅代表针对于一个 SQL 语句查询执行计划中的一部分查询的执行过程, 只是用来对查询执行计划进行管理和建模。Stage 在逻辑上又被分为一系列的 Task, 这些 Task 则是需要实际运行在 Presto

的各个 Worker 节点上的。

在 Presto 集群中，一个查询执行被分解成具有层级关系的一系列的 Stage，一个 Stage 又被拆分为一系列的 Task。每个 Task 处理一个或者多个 Split。每个 Task 都有对应的输入和输出。一个 Stage 被分解为多个 Task，从而可以并行地执行一个 Stage。Task 也采用了相同的机制：一个 Task 也可以被分解为一个或者多个 Driver，从而并行地执行一个 Task。

说明

由于一个 SQL 查询可以被分解为多个前后关联的 Stage，而每个 Stage 中均含有一个或者多个 Task，在这里我们约定：按照数据的流向，越靠近数据源的 Task 越处于上游，越远离数据源的 Task 越处于下游。

6. Driver

一个 Task 包含一个或者多个 Driver。一个 Driver 其实就是作用于一个 Split 的一系列 Operator 的集合。因此一个 Driver 用于处理一个 Split，并且生成相应的输出，这些输出由 Task 收集并且传送给其下游 Stage 中的一个 Task。一个 Driver 拥有一个输入和一个输出。

7. Operator

一个 Operator 代表对一个 Split 的一种操作，例如过滤、加权、转换等。一个 Operator 依次读取一个 Split 中的数据，将 Operator 所代表的计算和操作作用于 Split 的数据上，并产生输出。每个 Operator 均会以 Page 为最小处理单位分别读取输入数据和产生输出数据。Operator 每次只会读取一个 Page 对象，相应地，每次也只会产生一个 Page 对象。

8. Split

Split 即分片。一个分片其实就是一个大的数据集中的一个小子集。而 Driver 则是作用于一个分片上的一系列操作的集合，而每个节点上运行的 Task，又包含多个 Driver，从而一个 Task 可以处理多个 Split。其中每一种操作均由一个 Operator 表示。分布式查询执行计划的源 Stage (Source Stage) 通过 Connector 从数据源获得多个分片。Source Stage 对 Split 处理完毕之后，会将输出传递给其下游 Stage (通常其下游 Stage 的类型为 Fixed 或者 Single)。

当 Presto 执行一个查询的时候，首先会从 Coordinator 得到一个表对应的所有 Split。然后 Presto 就会根据查询执行计划，选择合适的节点运行相应的 Task 处理 Split。

9. Page

Page 是 Presto 中处理的最小数据单元。一个 Page 对象包含多个 Block 对象，而每个 Block 对象是一个字节数组，存储一个字段的若干行。多个 Block 横切的一行是真实的一行数据。一个 Page 最大为 1MB，最多 16×1024 行数据。Page 的结构如图 1-1 所示。

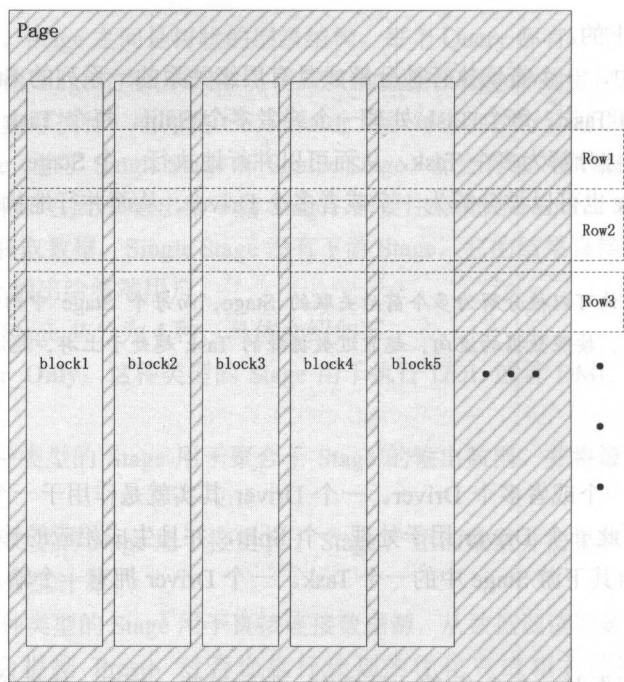


图 1-1

综上所述，Presto 执行查询的模型关系如图 1-2 所示。

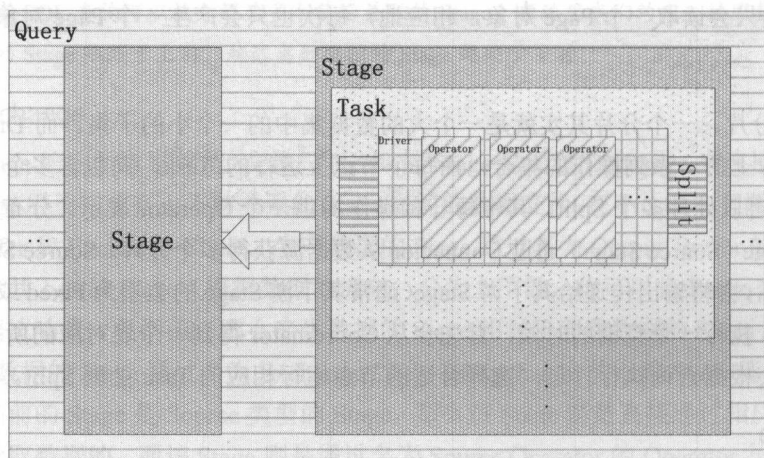


图 1-2

从图 1-2 中可以看到，在 Presto 中一次查询执行会被分解为多个 Stage，Stage 与 Stage 之间是有前后依赖关系的。每个 Stage 内部又会进一步地被分解为多个 Task，属于每个 Stage 的 Task 被均分在每个 Worker 上并行执行。在每个 Task 内部又会被分解为多个 Driver，每个 Driver 负责处理一个 Split，而且每个 Driver 由一系列前后相连的 Operator 组成，这里的每个 Operator 都代表针对一个 Split 的一种操作。

1.4 Presto 整体架构

通过上面的讲解，我们已经对 Presto 的发展历程、特点，以及 Presto 中的各种组件、概念和模型有了深入、细致的了解，在本节会详细讲解 Presto 的整体架构。

1.4.1 硬件架构

Presto 是一个完全基于内存的分布式查询执行引擎，因此 Presto 集群的硬件选择必须满足大内存、万兆网络和高计算能力的特点。由于 Presto 集群中的服务主要分为两种：Coordinator 和 Worker，因此 Presto 集群是 Master-Slave 的拓扑结构。除此之外，想要成功地使用 Presto，还需要具有客户端。

CLI 客户端：部署了 Presto 命令行客户端的服务器。

应用客户端：开发人员可以使用 Presto 的 JDBC 驱动程序，通过 Java 代码使用 Presto 进行大数据查询和计算。

综上所述，整个 Presto 集群的硬件包含 4 种服务器：Coordinator、Worker、Cli 客户端、应用客户端，其硬件架构如图 1-3 所示。

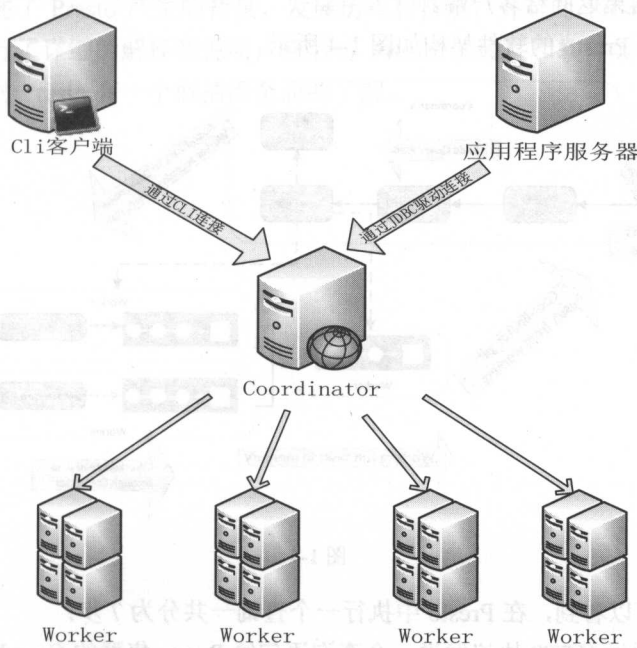


图 1-3

1.4.2 软件架构

在 1.4.1 节已经讲解了 Presto 的硬件架构，在本节会详细讲解 Presto 的软件架构及 Presto 查询执行的步骤。Presto 的软件架构说明如下。

- Presto 对多数据源的支持、数据源解耦以及易于扩展性归功于 Presto 的 Connector 设计。在 Presto 中，每种类型的数据源对应一种类型的 Connector，Presto 提供了 SPI，Presto 的 Developer 只需要实现 SPI 中提供的接口，就可以开发出从期望的数据源中读取数据的 Connector。
- Presto 是完全基于内存的分布式大数据查询执行引擎，所有的查询和计算均在内存中执行。
- Presto 基于 PipeLine 进行设计，每一次的查询都会被分解为多个分布在各个 Worker 上的 Task，每个 Task 都会与其上下游 Stage 中的 Task 在数据流上存在前后依赖关系，每个 Task 又会被进一步细分为多个 Driver，每个 Driver 由一系列有前后关系的 Operator 组成，每种 Operator 代表作用于一个 Split 上的一种操作。每当查询启动的时候，就会在每个 Worker 上启动相应的 Task，每个 Task 都会处理一个或者多个 Split，并且每处理一个 Buffer 大小的数据量时，就会将结果传输到下游 Stage 的 Task 中，这样基本可以保证数据实时动态地传输。
- Presto 客户端提交查询语句至 Coordinator，Coordinator 将查询最终分解为 Task，并交由各个 Worker 执行，Coordinator 从 Single Stage 中的 Task 获得最终的查询结果，并将查询结果返回给客户端。
- 综上所述，Presto 的软件架构如图 1-4 所示。

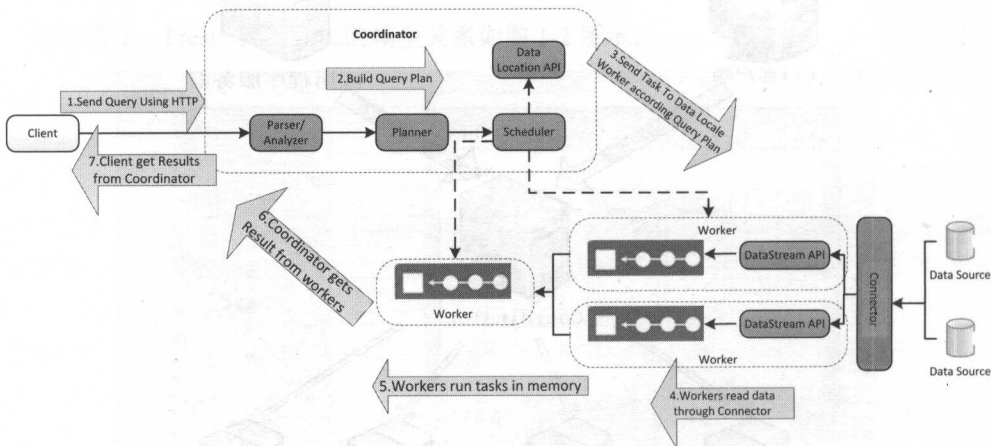


图 1-4

从图 1-4 中可以看到，在 Presto 中执行一个查询一共分为 7 步：

（1）客户端通过 HTTP 协议发送一个查询语句给 Presto 集群的 Coordinator。

（2）Coordinator 接到客户端传递过来的查询语句，会对该查询语句进行解析，生成查询执行计划，并根据查询执行计划依次生成 `SqlQueryExecution`、`SqlStageExecution`、`HttpRemoteTask`。Coordinator 会根据数据本地性生成对应的 `HttpRemoteTask`。

（3）Coordinator 将每个 Task 都分发到其所需要处理的数据所在的 Worker 上进行执行。这个过程是通过 `HttpRemoteTask` 中的 `HttpClient` 将创建或者更新 Task 的请求发送给数据所

在节点上 TaskResource 所提供的 RestFul 接口, TaskResource 接收到请求之后最终会在对应的 Worker 上启动一个 SqlTaskExecution 对象或者更新对应的 SqlTaskExecution 对象需要处理的 Split。

(4) 执行处于上游的 Source Stage 中的 Task, 这些 task 通过各种 Connector 从相应的数据源中读取所需要的数据。

(5) 处于下游 Stage 中的 Task 会读取上游 Stage 产生的输出结果, 并在该 Stage 每个 Task 所在 Worker 的内存中进行后续的计算和处理。

(6) Coordinator 从分发 Task 之后, 就会一直持续不断地从 Single Stage 中的 Task 获取计算结果, 并将计算结果缓存到 Buffer 中, 直到所有计算结束。

(7) Client 从提交查询语句之后, 就会不停地从 Coordinator 中获取本次查询的计算结果, 直到获得了所有的计算结果。并不是等到所有的查询结果都产生完毕之后一次全部显示出来, 而是每产生一部分, 就会显示一部分, 直到所有的查询结果都显示完毕。

1.5 小结

本章着重讲述了 Presto 产生的背景、发展历程和特点, 并且对 Presto 中的一些基本概念和软件术语进行了详细的解释和说明, 最后对 Presto 的硬件和软件架构进行了说明。通过本章读者可以对 Presto 有一个概括而全面的了解。

第2章

Presto 安装与部署

Presto 目前只能部署在 Linux 操作系统中，因此本章讲述的所有内容都是基于 Linux 操作系统并且以 Cluster 模式进行的 Presto 部署。

Presto 支持多种数据源，但是由于篇幅有限，本章不会对每种数据源都详细说明 Presto 的部署步骤，而是针对于 Hive 数据源，对 Presto 的部署步骤进行说明。

2.1 环境说明

想要以集群的方式进行 Presto 部署需要准备多台服务器，Presto 集群的环境如表 2-1 所示。

表 2-1

部署模式	集 群	
软件版本	Presto-0.107 Hadoop 2.5.0-cdh5.3.0 Hive 0.13.1-cdh5.3.0	
集群说明	一共选择 5 个节点，1 个节点作为 Coordinator，4 个节点作为 Worker。	
节点部署说明		
IP 地址	Host 名称	部署服务
192.168.100.10	BJ-PRESTO-TEST-100010.jd.com	Presto(Coordinator)、HiveMetaStore、Mysql(Hive 元数据库)
192.168.100.11	BJ-PRESTO-TEST-100011.jd.com	Presto(Worker)
192.168.100.12	BJ-PRESTO-TEST-100012.jd.com	Presto(Worker)
192.168.100.13	BJ-PRESTO-TEST-100013.jd.com	Presto(Worker)
192.168.100.14	BJ-PRESTO-TEST-100014.jd.com	Presto(Worker)

2.2 准备工作

为了方便 Presto 集群的部署, 首先选定 Coordinator 作为集群部署时的分发和控制节点, 因此就需要打通从 Coordinator 到各个 Worker 节点的 ssh 信任关系, 以方便后续的 Presto 集群管理和维护。

2.2.1 建立 SSH 信任关系

本节中所描述的 SSH 信任关系是指 Linux 操作系统之间的 SSH 信任关系。

1. 实现原理

使用“公私钥”认证的方式来进行 SSH 登录。“公私钥”认证方式如下:

假如有两台服务器: server1 和 server2, 我们需要建立从 server1 到 server2 的 SSH 信任关系。首先需要在 server1 上创建一对公私钥 (公钥文件路径: `~/.ssh/id_rsa.pub`; 私钥文件路径: `~/.ssh/id_rsa`), 然后将 server1 上公钥的内容追加到 server2 上的文件: `~/.ssh/authorized_keys` 的末尾 (注意: 不要在 server2 的文件: `~/.ssh/authorized_keys` 的最后一行直接追加, 而是在最后另起一行), 在 server1 上保留好私钥。当从 server1 通过 SSH 登录到 server2 时, SSH 程序会将 server1 上的私钥和 server2 上的公钥做匹配。如果匹配成功, 则不需要输入密码便可以登录了。

2. 环境说明

server1: 192.168.100.10

Server2: 192.168.100.11

3. 建立 SSH 信任

(1) 在 server1 上生成证书

在 server1 的 Presto 用户下执行 `ssh-keygen` 命令, 在需要输入的地方, 直接输入回车符, 生成建立安全信任关系的证书。执行结果如下:

```
[presto@BJ-PRESTO-TEST-100010 ~]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/presto/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/presto/.ssh/id_rsa.
Your public key has been saved in /home/presto/.ssh/id_rsa.pub.
The key fingerprint is:
49:cf:48:83:2c:25:62:8b:9c:0d:90:67:75:22:f6:8a
presto@BJ-PRESTO-TEST-100010.jd.com
The key's randomart image is:
+--[ RSA 2048 ]-----+
|oo=.+ o              |
|o+*= * .            |
|.=..o o +           |
```

```

| . . . o * |
| E .   S o |
|           |
|           |
|           |
|           |
+-----+
[presto@BJ-PRESTO-TEST-100010 ~]$

```

注意：在程序提示输入 passphrase 时直接输入回车符，表示无证书密码。

上述命令将生成私钥证书 id_rsa 和公钥证书 id_rsa.pub，存放在当前登录用户的根目录下的.ssh 子目录中。

(2) 查看生成的公私钥证书

生成的公钥证书、私钥证书存放在目录 ~/.ssh 目录下，生成的文件如下：

```

[presto@BJ-PRESTO-TEST-100010 ~]$ cd ~/.ssh
[presto@BJ-PRESTO-TEST-100010.ssh]$ ls -l
total 16
-rw-----. 1 presto presto 2448 Aug 13 12:34 authorized_keys
-rw-----. 1 presto presto 1675 Aug 28 14:50 id_rsa
-rw-r--r--. 1 presto presto 415 Aug 28 14:50 id_rsa.pub
-rw-r--r--. 1 presto presto 1584 Aug 28 14:49 known_hosts
[presto@BJ-PRESTO-TEST-100010 .ssh]$

```

(3) 建立 server1 到 server2 的 SSH 信任关系

将 server1 上的公钥证书 ~/.ssh/id_rsa.pub 中的内容添加到 server2 的 Presto 根目录的.ssh 子目录中 authorized_keys 文件的末尾，此时需要输入 server2 的 Presto 用户密码（还未建立信任关系）。建立了 server1 到 server2 的信任关系后，server1 就可以不用再输入密码了。建立 server1 到 server2 的 SSH 信任关系有两种方式：

第一种：复制 server1 的文件 ~/.ssh/ id_rsa.pub 中的内容，并将其追加到 server2 的文件 ~/.ssh/ authorized_keys 中的末尾。

第二种：在 server1 上执行命令：ssh-copy-id -i ~/.ssh/id_rsa.pub 192.168.100.11，完成信任关系的建立。

这里我们采用第二种方式，执行结果如下：

```

[presto@BJ-PRESTO-TEST-100010 .ssh]$ ssh-copy-id -i ~/.ssh/id_rsa.pub
192.168.100.11
presto@192.168.100.11's password: #这里需要输入 server2 的登录密码
Now try logging into the machine, with "ssh '192.168.100.11'", and check in:

    .ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
#检查是否可以无密码登录到 server2
[presto@BJ-PRESTO-TEST-100010 .ssh]$ ssh 192.168.100.11
Last login: Fri Aug 28 14:55:12 2015 from 192.168.100.10
[presto@BJ-PRESTO-TEST-100011 ~]$
#已经可以在 server1 上无密码登录到 server2 了

```


4. 验证

通过上面的操作已经建立起了 server1 到 server2 的 SSH 信任关系，下面对该信任关系进行验证如下：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ scp ~/file1 192.168.100.11:/tmp/
file1 100% 19 0.0KB/s 00:00
[presto@BJ-PRESTO-TEST-100010 ~]$ ssh 192.168.100.11 'ls /tmp/'
file1
[presto@BJ-PRESTO-TEST-100010 ~]$
```

从上面的验证结果可以看出，目前可以从 server1 无密码地向 server2 传递文件，并且可以无密码地在 server2 上执行远程命令。

5. 自动化脚本

前面介绍了如何建立从 Coordinator 到一个 Worker 上的 SSH 信任关系，但是我们在进行 Presto 集群建设的时候，集群中的节点数一般会比较，若对每个节点都通过手动的方式建立一次从 Coordinator 到该节点的信任关系，则势必会进行非常烦琐的重复性操作，因此，可以结合 expect 在 Coordinator 上使用 shell 脚本一次性自动化完成整个集群 SSH 信任关系的建立。

首先需要创建一个文件 servers.txt，在该文件中记录了集群中所有服务器的 IP 地址，其内容如下：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat servers.txt
192.168.100.10
192.168.100.11
192.168.100.12
192.168.100.13
192.168.100.14
[presto@BJ-PRESTO-TEST-100010 ~]$
```

其次创建一个 shell 脚本：establishSSH.sh，用于自动完成 Coordinator 到各个服务器 SSH 信任关系的建立，该脚本的内容如下：

```
#!/bin/bash
#指定需要建立信任关系的用户名
USER_UID=presto
#系统用户的家目录
USER_DIR=/home/presto
#用户对应的密码，从这里可以看出，该脚本要求每台服务器的相同用户的密码必须一致
IP_PW=PASSWORD
#定义 function establishSSH
establishSSH ()
{
cat >establishSSH.exp<<EOF
#!/usr/bin/expect
spawn ssh-copy-id -i $USER_DIR/.ssh/id_rsa.pub $SSH_IP
expect {
"*yes/no*" {send "yes\r"; exp_continue}
```

```

"*password*" {send "$IP_PW\r";}
}
expect eof
EOF
chmod 755 establishSSH.exp
./establishSSH.exp > /dev/null
/bin/rm -rf establishSSH.exp
}

#####main#####

if [ -f servers.txt ]
then
:
else
echo
echo "-----Please touch File: servers.txt -----"
echo "##### servers.txt For Example #####"
echo "192.168.100.10"
exit 0
fi

if ( rpm -qa | grep -q expect )
then
:
else
yum -y install expect > /dev/null
fi

if [ -f $USER_DIR/.ssh/id_rsa.pub ]
then
:
else
ssh-keygen
fi

for SSH_IP in `servers.txt`
do
establishSSH
if [ $? -eq 0 ]
then
echo "-----$SSH_IP is OK-----"
else
echo "-----$SSH_IP is failed-----"
fi
done

```

通过在服务器 192.168.100.10 的目录/tmp/下建立两个文件 servers.txt 和 establishSSH.sh, 更改 establishSSH.sh 文件的权限为 755, 然后执行 establishSSH.sh, 即可建立起 Coordinator 到各个 Worker 的 SSH 信任关系。

2.2.2 安装 Java

由于 Presto-0.107 必须运行在 JDK 1.8 以上，因此我们选择 JDK 1.8 作为 Presto 的 Java 运行环境。JDK 的二进制包可以从 Oracle 提供的 Java 官方网站中下载，在这里就不再赘述了。本书使用的 JDK 版本为：jdk1.8.0_40。下载完 jdk1.8.0_40 的二进制包之后，按照如下步骤执行。

(1) 解压

```
[presto@BJ-PRESTO-TEST-100010 ~]$ tar zxvf jdk1.8.0_40.tar.gz -C /software/servers/
```

(2) 配置 JAVA_HOME 和 Path

在系统配置文件 ~/.bashrc 中添加以下配置项：

```
export JAVA_HOME=/software/servers/jdk1.8.0_40
export PATH=$PATH:$JAVA_HOME/bin
```

然后执行命令：source ~/.bashrc，使配置生效。

(3) 验证

```
[presto@BJ-PRESTO-TEST-100010 ~]$ java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
[presto@BJ-PRESTO-TEST-100010 ~]$
```

输入命令：java -version，出现上面的信息，则表示 JDK 安装成功。

2.2.3 安装 Maven

由于使用源码编译的方式进行 Presto 部署，而 Presto 使用 Maven 进行编译，所以本章介绍如何安装 Maven。

由于 Presto 的编译需要使用 Maven-3.2.3 以上版本，因此我们使用 Maven-3.2.5 版本。从 Maven 官方网站下载 Maven-3.2.5 二进制包之后，按照如下步骤进行安装。

(1) 解压

```
[presto@BJ-PRESTO-TEST-100010 ~]$ tar zxvf apache-maven-3.2.5-bin.tar.gz -C /software/servers/
```

(2) 配置 MAVEN_HOME 和 Path

在系统配置文件 ~/.bashrc 中添加以下配置项：

```
export MAVEN_HOME=/software/servers/apache-maven-3.2.5
export PATH=$MAVEN_HOME/bin:$PATH
```

然后执行命令：source ~/.bashrc，使配置生效。

(3) 验证

```
[presto@BJ-PRESTO-TEST-100010 ~]$ mvn -version
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8ceal;
```



```
2014-12-15T01:29:23+08:00)
```

```
Maven home: /software/servers/apache-maven-3.2.5
Java version: 1.8.0_40-ea, vendor: Oracle Corporation
Java home: /software/servers/jdk1.8.0_40/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "linux", version: "2.6.32-431.el6.x86_64", arch: "amd64", family: "unix"
[presto@BJ-PRESTO-TEST-100010 ~]$
```

输入命令：`mvn -version`，出现上面的信息，则表示 Maven 安装成功。

2.2.4 安装 Hive

Presto 在访问 Hive 数据仓库中的数据时需要得到 Hive 中的所有元数据信息，因此需要部署一个 HiveMetaStore 服务提供 Hive 的元数据信息，本节会详细讲解如何部署 HiveMetaStore。

从 Cloudera 官方网站下载 Hive 0.13.1-cdh5.3.0 二进制包，然后按照如下步骤进行安装。

1. 解压

```
[presto@BJ-PRESTO-TEST-100010 ~]$ tar zxvf hive-0.13.1-cdh5.3.0.tar.gz -C
/software/servers/
```

2. 配置 HIVE_HOME 和 Path

在系统配置文件 `~/.bashrc` 中添加以下配置项：

```
export HIVE_HOME=/software/servers/hive-0.13.1-cdh5.3.0
export PATH=$PATH:$HIVE_HOME/bin
```

然后执行命令：`source ~/.bashrc`，使配置生效。

3. 验证

```
[presto@BJ-PRESTO-TEST-100010 ~]$ hive --version
Hive 0.13.1-cdh5.3.0
Subversion
git://localhost/var/lib/jenkins/workspace/generic-binary-tarball-
and-maven-deploy/CDH5.3.0-Packaging-Hive-2014-12-16_19-00-17/hive-0.13.1-cdh5.3.0
-r 8e266e052e423af592871e2dfe09d54c03f6a0e8
Compiled by jenkins on Tue Dec 16 19:03:25 PST 2014
From source with checksum d065deb124f4d43fc970e0b8fd659236
[presto@BJ-PRESTO-TEST-100010 ~]$
```

输入命令：`hive -version`，出现上面的信息，则表示 Hive 安装成功。

4. 配置 hiveMetaStore

修改 Hive 的配置文件：`$HIVE_HOME/conf/hive-site.xml`，进行如下配置：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat
/software/servers/hive-0.13.1-cdh5.3.0/conf/hive-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
```

```
<configuration>
  <property>
    <name>hive.hwi.war.file</name>
    <value>lib/hive-hwi-0.13.1-cdh5.3.0.jar</value>
  </property>

  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://
BJ-PRESTO-TEST-100010.JD.COM:3306/test_presto_hive?characterEncoding=utf-8&autoReconnect=true&maxReconnects=5&connectTimeout=10000</value>
  </property>

  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>hdfs://namespacel/user/presto</value>
    <final>true</final>
  </property>

  <property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.jdbc.Driver</value>
  </property>

  <property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>presto</value>
  </property>

  <property>
    <name>javax.jdo.option.ConnectionPassword</name>
    <value>presto-password</value>
  </property>

  <property>
    <name>datanucleus.fixedDatastore</name>
    <value>false</value>
  </property>

  <property>
    <name>datanucleus.autoCreateSchema</name>
    <value>true</value>
  </property>

  <property>
    <name>datanucleus.autoCreateTables</name>
    <value>true</value>
  </property>

  <property>
    <name>datanucleus.autoCreateColumns</name>
    <value>true</value>
  </property>
</configuration>
```

```
</property>
</configuration>
```

对上述配置文件的配置项说明如下。

javax.jdo.option.ConnectionURL: 用于存储 Hive 元数据的 MySQL 的连接字符串。其中的 test_presto_hive 是 MySQL 中的库名。

hive.metastore.warehouse.dir: Hive 数据仓库中的实际数据在 HDFS 上的存储路径。

javax.jdo.option.ConnectionDriverName: 元数据库对应的驱动类，由于我们选用 MySQL 数据库作为 Hive 数据仓库中的元数据库，因此上述配置文件中配置的就是 MySQL 的驱动类。

javax.jdo.option.ConnectionUserName: 连接 MySQL 数据库的用户名。

javax.jdo.option.ConnectionPassword: 连接 MySQL 数据库的密码。

datanucleus.autoCreateTables: 当所需要的表在 MySQL 的库中不存在时，是否自动创建。

datanucleus.autoCreateColumns: 当所需要的列在 MySQL 的表中不存在时，是否自动创建。

datanucleus.fixedDatastore: 在 Hive 与 MySQL 进行交互时，是否固定 MySQL 中的元数据库，不进行元数据结构检查，也不会自动添加不存在的元数据库的 schema、table 和 column。当某个数据仓库刚开始投入使用的时候，由于很多元数据的数据结构还没有固定，建议将该配置项设置为 false，表示不固定 MySQL 中的元数据库；当某个数据仓库已经运行了很长一段时间，则建议将该配置项设置为 true，表示固定 MySQL 中的元数据库，不进行元数据结构检查，这样可以提高查询效率。

datanucleus.autoCreateSchema: 当相应的 schema 在 MySQL 实例中不存在时，是否自动创建。

说明

对于配置项 datanucleus.autoCreateTables、datanucleus.autoCreateColumns、datanucleus.fixedDatastore 和 datanucleus.autoCreateSchema，在刚开始部署 hiveMetaStore 服务的时候，这些配置项的值如上述配置文件所示。这样可以保证当 MySQL 中缺失某些 schema、table 或者 column 的时候，可以自动进行创建。但是这样的配置会导致 HiveMetaStore 的性能低下，因此在 HiveMetaStore 无故障运行一段时间之后（建议 1 个星期），MySQL 中不会再出现缺失的 schema、table 和 column 的时候，重新设置这 4 个配置项的值，来提升 HiveMetaStore 的性能，重新设置的值如下所示：

```
<property>
  <name>datanucleus.fixedDatastore</name>
  <value>true</value>
</property>

<property>
  <name>datanucleus.autoCreateSchema</name>
  <value>>false</value>
</property>
```



```

<property>
  <name>datanucleus.autoCreateTables</name>
  <value>>false</value>
</property>

<property>
  <name>datanucleus.autoCreateColumns</name>
  <value>>false</value>
</property>

```

5. 启动 HiveMetaStore 服务

启动 HiveMetaStore 服务的命令如下：

```

[presto@BJ-PRESTO-TEST-100010 hive-0.13.1-cdh5.3.0]$ nohup hive --service
metastore -p 9083 >/data1/hive_tmp/presto_metastore_9083.log 2>&1 &

```

上述启动命令的含义为：在后台启动 Hive 的 metastore 服务，metastore 服务监听 9083 端口，并将错误和常规日志输出到日志文件/data1/hive_tmp/presto_metastore_9083.log 中。

2.3 源码编译

有两种方式可以获得安装包：直接下载和源码编译。

我们可以从 Presto 的 GitHub 上下载源码自己动手编译成 tar 包，也可以从官方网站中下载已经编译好的安装包：<https://github.com/facebook/presto/archive/0.107.tar.gz>。下面介绍如何使用源码手动编译安装包，然后进行安装。

2.3.1 下载源码

首先登录到 Coordinator 服务器，从 GitHub 上下载 Presto 最新源码，下载地址为：<https://github.com/facebook/presto.git>。然后将最新源码切换至 Presto-0.107 版本，如下所示。

```

[presto@BJ-PRESTO-TEST-100010 lvxin]$ cd lvxin-test/
[presto@BJ-PRESTO-TEST-100010 lvxin-test]$ git clone
https://github.com/facebook/presto.git
Initialized empty Git repository in
/home/presto/lvxin/lvxin-test/presto/.git/
remote: Counting objects: 136786, done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 136786 (delta 10), reused 3 (delta 3), pack-reused 136767
Receiving objects: 100% (136786/136786), 45.66 MiB | 320 KiB/s, done.
Resolving deltas: 100% (66026/66026), done.
[presto@BJ-PRESTO-TEST-100010 lvxin-test]$ ll
总用量 4
drwxr-x--- 36 presto presto 4096 8月 31 10:45 presto
[presto@BJ-PRESTO-TEST-100010 lvxin-test]$ cd presto/
[presto@BJ-PRESTO-TEST-100010 presto]$ git checkout tags/0.107

```


2.3.2 源码结构说明

进入到 Presto 的源码的根目录，查看其所有子工程，内容如下：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cd lvxin/lvxin-test/presto/
[presto@BJ-PRESTO-TEST-100010 presto]$ ls -l
总用量 172
-rw-r----- 1 presto presto 611 8月 31 10:45 CONTRIBUTING.md
-rw-r----- 1 presto presto 11358 8月 31 10:45 LICENSE
-rw-r----- 1 presto presto 35599 8月 31 11:00 pom.xml
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-base-jdbc
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-benchmark
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-benchmark-driver
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-cassandra
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-cli
drwxr-x--- 4 presto presto 4096 8月 31 11:26 presto-client
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-docs
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-example-http
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-hive
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-hive-cdh4
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-hive-cdh5
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-hive-hadoop1
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-hive-hadoop2
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-jdbc
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-kafka
drwxr-x--- 5 presto presto 4096 8月 31 11:07 presto-main
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-ml
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-mysql
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-orc
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-parser
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-postgresql
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-raptor
drwxr-x--- 4 presto presto 4096 8月 31 11:26 presto-server
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-spi
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-tests
drwxr-x--- 4 presto presto 4096 8月 31 11:07 presto-tpch
drwxr-x--- 4 presto presto 4096 8月 31 11:08 presto-verifier
-rw-r----- 1 presto presto 3312 8月 31 11:00 README.md
drwxr-x--- 4 presto presto 4096 8月 31 10:45 src
```

下面对 Presto 源码根目录下的各个子工程以及相关文件进行说明。

pom.xml: presto-root 工程的 pom 文件。Presto 的所有子工程都隶属于一个统一的 presto-root 工程，该工程是一个虚拟工程，用于组织所有的子工程，该 pom 文件描述了 presto-root 工程的所有信息，以及其所有的子工程，该 pom 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```

<!--presto 中各个模块之间的依赖注入、配置管理等均依赖于 airlift 工程-->
<parent>
  <groupId>io.airlift</groupId>
  <artifactId>airbase</artifactId>
  <version>39</version>
</parent>
<!--定义 presto-root 工程详细信息 -->
<groupId>com.facebook.presto</groupId>
<artifactId>presto-root</artifactId>
<version>0.107</version>
<packaging>pom</packaging>

<name>presto-root</name>
<description>Presto</description>
<url>https://github.com/facebook/presto</url>
  <!--定义 git 详细信息-->
  <scm>
    <connection>scm:git:git://github.com/facebook/presto.git</connection>
    <url>https://github.com/facebook/presto</url>
    <tag>0.107</tag>
  </scm>
  <!--定义 maven 相关属性 -->
  <properties>
    <air.main.basedir>${project.basedir}</air.main.basedir>
    <air.check.skip-extended>true</air.check.skip-extended>
    <air.check.skip-license>false</air.check.skip-license>
    <air.check.fail-checkstyle>true</air.check.fail-checkstyle>
    <air.check.skip-checkstyle>false</air.check.skip-checkstyle>
    <depantlr.version>4.3</depantlr.version>
    <dep.airlift.version>0.110</dep.airlift.version>
    <dep.packaging.version>${dep.airlift.version}</dep.packaging.version>
    <dep.slice.version>0.14</dep.slice.version>
    <cli.skip-execute>true</cli.skip-execute>
    <cli.main-class>None</cli.main-class>
    <!-- use a fractional hour timezone offset for tests -->
    <air.test.timezone>Asia/Katmandu</air.test.timezone>
    <air.test.parallel>methods</air.test.parallel>
    <air.test.thread-count>4</air.test.thread-count>
    <air.test.jvmsize>1792m</air.test.jvmsize>
    <air.javadoc.lint>-missing</air.javadoc.lint>
  </properties>
  <!--指定 presto-root 工程所包含的子工程-->
  <modules>
    <module>presto-spi</module>
    <module>presto-kafka</module>
    <module>presto-cassandra</module>
    <module>presto-orc</module>
    <module>presto-hive</module>
    <module>presto-hive-hadoop1</module>
    <module>presto-hive-hadoop2</module>
    <module>presto-hive-cdh4</module>
  </modules>

```

```

<module>presto-hive-cdh5</module>
<module>presto-example-http</module>
<module>presto-tpch</module>
<module>presto-raptor</module>
<module>presto-base-jdbc</module>
<module>presto-mysql</module>
<module>presto-postgresql</module>
<module>presto-client</module>
<module>presto-parser</module>
<module>presto-main</module>
<module>presto-ml</module>
<module>presto-benchmark</module>
<module>presto-tests</module>
<module>presto-jdbc</module>
<module>presto-cli</module>
<module>presto-benchmark-driver</module>
<module>presto-server</module>
<module>presto-docs</module>
<module>presto-verifier</module>
</modules>
<!--presto-root 工程的依赖管理-->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.facebook.presto</groupId>
      <artifactId>presto-spi</artifactId>
      <version>${project.version}</version>
    </dependency>
    .....
    <dependency>
      <groupId>org.jgrapht</groupId>
      <artifactId>jgrapht-core</artifactId>
      <version>0.9.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>

.....
</project>

```

从 pom.xml 文件的内容中可以看出 presto-root 工程的所有子工程都是在元素 `<modules></modules>` 中进行定义的。

presto-base-jdbc: 该子工程主要提供访问各种 RDBMS 数据源数据的途径，该工程定义了访问 RDBMS 数据源的接口和基类，若想要访问 RDBMS 数据源中的数据，则需要实现该工程中的接口或者扩展相关的基类。例如 presto-mysql 子工程就需要依赖于该工程来访问 MySQL 中的数据。

presto-cassandra: 该子工程主要提供对 Cassandra 中数据的查询和分析。

presto-cli: 该工程提供通过命令行访问 Presto 集群的途径。

presto-client: 该工程提供提交查询时使用的客户端、Session 参数等。

presto-docs: 该工程生成 Presto 的相关文档。

presto-example-http: 该工程用于读取通过 HTTP 发送的以逗号分隔的数据。除此之外，该工程也作为一个 example 工程，说明一个新的 Connector 如何开发。因此，若读者想要开发一个读取新的数据源的 Connector，则可以参照该工程。

presto-hive: 该工程提供了读取 Hive 数据仓库中的数据的 Connector。该工程作为一个基础工程，会被读取不同版本 Hive 数据仓库中数据的 Connector 所依赖。

presto-hive-cdh4: 该工程提供了读取 CDH4 版本的 Hive 数据仓库中的数据的 Connector，该工程依赖于工程 presto-hive。

presto-hive-cdh5: 该工程提供了读取 CHD5 版本的 Hive 数据仓库中的数据的 Connector，该工程依赖于工程 presto-hive。

presto-hive-hadoop1: 该工程提供了读取 apache-hadoop 1 版本的 Hive 数据仓库中的数据的 Connector，该工程依赖于工程 presto-hive。

presto-hive-hadoop2: 该工程提供了读取 apache-hadoop 2 版本的 Hive 数据仓库中的数据的 Connector，该工程依赖于工程 presto-hive。

presto-jdbc: 该工程提供了 presto jdbc 驱动，Java 程序可以通过该 JDBC 驱动访问 Presto 集群，进行数据查询和计算。

presto-kafka: 该工程提供了访问 kafka 中数据的 Connector，通过 presto-kafka 可以将 kafka 中的每个 topic 作为一个 table 进行查询和分析。

presto-server: 该工程中没有任何源码，唯一的一个文件就是 presto.xml，在该文件中说明了各个子工程的依赖关系，以及各个子工程生成的 jar 包在打包时存放的路径。该工程的唯一作用就是将 Presto 的各个子工程都组织起来，从而在进行源码编译的时候，将 Presto 的各个子工程产生的 jar 包，以指定的目录结构都打包到一个 tar 包中。

presto-ml: 使用该工程提供的 Java 包可以进行机器学习计算，但是目前该工程还没有进行实现。

presto-mysql: 该工程提供访问 MySQL 的 Connector。

presto-orc: 该工程进行了 ORC 文件读取的优化。

presto-parser: 该工程提供了 SQL 语句的文法和语法分析。

presto-postgresql: 该工程提供访问 PostgreSQL 的 Connector。

presto-raptor: 该工程提供访问每台节点操作系统中数据的 Connector。

presto-main: 该工程是 Presto 最主要的一个子工程，提供了 Presto 集群中各个节点上的 Presto 服务启动所需要的 jar 包。

presto-spi: SPI (Service Provider Interface)，该子工程是面向模块工程的体现，该工程中提供了实现各种数据源 Connector 所需要实现的所有接口。若想要开发一个读取一种新类型数据源的 Connector，就需要在新 Connector 对应的子工程中实现 SPI 提供的各种接口。

presto-tests: 该工程提供了 Presto 中的各种测试类，若进行 Presto 功能性开发或者 Bug 修复，则需要在该工程中书写相应的测试类，只有相关的测试类运行通过之后才能证明开发的功能没有 Bug，或者相应的 Bug 已经被修复。

2.3.3 编译

完成了源码的下载之后，直接进入 Presto 源码的根目录，通过 Maven 命令执行源码的编译，命令如下：

```
[presto@BJ-PRESTO-TEST-100010 presto]$ mvn -T2C install -DskipTests
.....
[INFO] Installing
/home/presto/lvxin/lvxin-test/presto/presto-server/target/presto-server-0.107.tar.gz
to
/data11/m2/repo/com/facebook/presto/presto-server/0.107/presto-server-0.107.tar.gz
[INFO] Installing
/home/presto/lvxin/lvxin-test/presto/presto-server/pom.xml to
/data11/m2/repo/com/facebook/presto/presto-server/0.107/presto-server-0.107.pom
[INFO] Installing
/home/presto/lvxin/lvxin-test/presto/presto-server/target/presto-server-0.107.jar
to
/data11/m2/repo/com/facebook/presto/presto-server/0.107/presto-server-0.107.jar
[INFO]
-----
[INFO] Reactor Summary:
[INFO]
[INFO] presto-root ..... SUCCESS [ 3.673 s]
[INFO] presto-spi ..... SUCCESS [ 9.518 s]
[INFO] presto-client ..... SUCCESS [ 3.075 s]
[INFO] presto-parser ..... SUCCESS [ 9.537 s]
[INFO] presto-tpch ..... SUCCESS [ 2.658 s]
[INFO] presto-main ..... SUCCESS [ 29.151 s]
[INFO] presto-tests ..... SUCCESS [ 7.774 s]
[INFO] presto-kafka ..... SUCCESS [ 9.250 s]
[INFO] presto-cassandra ..... SUCCESS [ 11.126 s]
[INFO] presto-orc ..... SUCCESS [ 8.442 s]
[INFO] presto-benchmark ..... SUCCESS [ 3.633 s]
[INFO] presto-hive ..... SUCCESS [ 13.317 s]
[INFO] presto-hive-hadoop1 ..... SUCCESS [ 7.955 s]
[INFO] presto-hive-hadoop2 ..... SUCCESS [ 8.158 s]
[INFO] presto-hive-cdh4 ..... SUCCESS [ 8.274 s]
[INFO] presto-hive-cdh5 ..... SUCCESS [ 8.091 s]
[INFO] presto-example-http ..... SUCCESS [ 5.302 s]
[INFO] presto-raptor ..... SUCCESS [ 18.941 s]
[INFO] presto-base-jdbc ..... SUCCESS [ 6.533 s]
[INFO] presto-mysql ..... SUCCESS [ 10.951 s]
[INFO] presto-postgresql ..... SUCCESS [ 6.541 s]
[INFO] presto-ml ..... SUCCESS [ 9.559 s]
[INFO] presto-jdbc ..... SUCCESS [ 24.024 s]
[INFO] presto-cli ..... SUCCESS [ 6.766 s]
[INFO] presto-benchmark-driver ..... SUCCESS [ 6.417 s]
[INFO] presto-server ..... SUCCESS [ 30.690 s]
[INFO] presto-docs ..... SUCCESS [ 50.940 s]
[INFO] presto-verifier ..... SUCCESS [ 11.806 s]
[INFO]
-----
```



```
[INFO] BUILD SUCCESS
[INFO]
```

```
-----
[INFO] Total time: 01:46 min (Wall Clock)
[INFO] Finished at: 2015-08-31T11:09:02+08:00
[INFO] Final Memory: 451M/8229M
[INFO]
-----
```

出现以上信息,就表示已经成功完成了 Presto 的源码编译。对编译命令 `mvn -T2C install-DskipTests` 说明如下。

-T2C: 说明一个 CPU 核心启动两个线程进行编译,可以加快源码编译的速度。

install: 将编译完成的 jar 包直接安装到 Maven 库中。

-DskipTests: 跳过测试工程和测试类。

我们在进行 Presto 部署和使用的时候主要使用 3 个工程: `presto-server`, `presto-cli` 和 `presto-jdbc`, 这 3 个工程的详细信息说明如下。

presto-server: 编译该工程生成部署 Presto 集群所需要的所有二进制 tar 包, 该工程没有任何源码, 其唯一的作用就是组合 Presto 的其他各个工程, 并将各个子工程编译后的二进制 jar 包按照指定的路径结构统一打包为一个可用于部署 Presto 集群的 tar 包, 上面已经对 Presto 的所有工程进行了整体编译。编译完成后, `presto-server` 工程生成的 tar 包的路径为: `/home/presto/lvxin/lvxin-test/presto/presto-server/target/presto-server-0.107.tar.gz`。后续我们将会使用这个压缩包部署 Presto 集群。在 `presto-server` 工程的 `src/main/provisio/presto.xml` 文件中, 定义了需要将哪些 Presto 子工程的二进制包打包进 presto server 的 tar 包中, 并且指定了不同的子工程生成的 jar 包都存放在 tar 包的什么目录中。该文件的内容如下:

```
[presto@BJ-PRESTO-TEST-100010 presto]$ cat
presto-server/src/main/provisio/presto.xml
<runtime>
  <!-- Target -->
  <archive name="${project.artifactId}-${project.version}.tar.gz" />
  <!-- Notices -->
  <fileSet to="/">
    <directory path="${basedir}">
      <include>NOTICE</include>
      <include>README.txt</include>
    </directory>
  </fileSet>

  <!-- Launcher 生成 bin 脚本-->
  <artifactSet to="bin">
    <artifact
id="io.airlift:launcher:tar.gz:bin:${dep.packaging.version}">
      <unpack />
    </artifact>
    <artifact
id="io.airlift:launcher:tar.gz:properties:${dep.packaging.version}">
      <unpack filter="true" />
```

```

    </artifact>
</artifactSet>

<!--Server 定义 -->
<!--将 presto-main 生成的 jar 包放入到 tar 包的 lib 目录中 -->
<artifactSet to="lib">
    <artifact
id="${project.groupId}:presto-main:${project.version}" />
</artifactSet>
<!-- Plugins -->
<!--将 presto- cassandra 生成的 jar 包放入到 tar 包的 plugin/cassandra 目录中 -->
<artifactSet to="plugin/cassandra">
    <artifact id="${project.groupId}:presto-cassandra:zip:${project.
version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- example-http 生成的 jar 包放入到 tar 包的 plugin/example-http 目录中
-->
<artifactSet to="plugin/example-http">
    <artifact
id="${project.groupId}:presto-example-http:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- hive-cdh4 生成的 jar 包放入到 tar 包的 plugin/hive-cdh4 目录中 -->
<artifactSet to="plugin/hive-cdh4">
    <artifact
id="${project.groupId}:presto-hive-cdh4:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- hive-cdh5 生成的 jar 包放入到 tar 包的 plugin/hive-cdh5 目录中 -->
<artifactSet to="plugin/hive-cdh5">
    <artifact
id="${project.groupId}:presto-hive-cdh5:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- hive- hadoop1 生成的 jar 包放入到 tar 包的 plugin/hive- hadoop1 目录中
-->
<artifactSet to="plugin/hive-hadoop1">
    <artifact
id="${project.groupId}:presto-hive-hadoop1:zip:${project. version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- hive- hadoop2 生成的 jar 包放入到 tar 包的 plugin/hive- hadoop2 目录中
-->
<artifactSet to="plugin/hive-hadoop2">
    <artifact
id="${project.groupId}:presto-hive-hadoop2:zip:${project. version}">

```

```

        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- hive- kafka 生成的 jar 包放入到 tar 包的 plugin/kafka 目录中 -->
<artifactSet to="plugin/kafka">
    <artifact
id="${project.groupId}:presto-kafka:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- ml 生成的 jar 包放入到 tar 包的 plugin/ml 目录中 -->
<artifactSet to="plugin/ml">
    <artifact
id="${project.groupId}:presto-ml:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto-mysql 生成的 jar 包放入到 tar 包的 plugin/mysql 目录中 -->
<artifactSet to="plugin/mysql">
    <artifact
id="${project.groupId}:presto-mysql:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- postgresql 生成的 jar 包放入到 tar 包的 plugin/postgresql 目录中 -->
<artifactSet to="plugin/postgresql">
    <artifact
id="${project.groupId}:presto-postgresql:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- raptor 生成的 jar 包放入到 tar 包的 plugin/raptor 目录中 -->
<artifactSet to="plugin/raptor">
    <artifact
id="${project.groupId}:presto-raptor:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
<!--将 presto- tpch 生成的 jar 包放入到 tar 包的 plugin/tpch 目录中 -->
<artifactSet to="plugin/tpch">
    <artifact
id="${project.groupId}:presto-tpch:zip:${project.version}">
        <unpack />
    </artifact>
</artifactSet>
</runtime>

```

presto-cli: 编译该工程可以生成用于连接到 Presto 集群，并接收 SQL 和命令输入的 Presto 命令行客户端。该工程编译完成之后，会生成一个 jar 文件，运行该文件就会提供一个 Presto 命令行客户端。该文件的路径为：`/home/presto/lvxin/lvxin-test/presto/presto-cli/target/presto-cli-0.107-executable.jar`。关于该 jar 文件怎么使用，在 2.4.2 节会进行说明。

presto-jdbc:编译该工程可以生成 Presto 的 JDBC 驱动包, 应用程序可以通过该 JDBC 驱动连接到 Presto 集群, 从而进行数据查询和计算。最终生成的 JDBC 驱动文件的路径为: `/home/presto/lvxin/lvxin-test/presto/presto-jdbc/target/presto-jdbc-0.107.jar`。

2.4 部署

通过 2.2 节已经完成了 Coordinator 节点与各个 Worker 节点之间信任关系的建立, 并且在 2.3 节完成了源码的编译, 因此可以直接使用已经编译完成的安装包进行 Presto 集群部署。而 Presto 集群部署分为三部分: Presto 服务部署、Presto-cli 客户端部署和 Presto JDBC 使用。下面分别对这三部分进行详细说明。

2.4.1 服务部署

Presto 服务部署是指以集群的方式部署 Presto 服务。在进行 Presto 服务部署的时候我们只需要使用 `presto-server-0.107.tar.gz` 二进制包。首先将二进制压缩包同步到各个节点上并解压, 然后在各个节点上修改相应的配置项, 最后启动 Presto 集群。下面会以读取 Hive 数据源中的数据为例, 针对 Presto 服务部署的每一步进行讲解。

1. 文件同步

由于前面已经打通了 Presto 集群中 Coordinator 与各个 Worker 节点的 SSH 信任关系, 因此可以通过 Shell 脚本批量 scp 并解压压缩包: `/home/presto/lvxin/lvxin-test/presto/presto-server/target/presto-server-0.107.tar.gz`, 具体操作步骤如下所示。

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat servers.txt
192.168.100.10
192.168.100.11
192.168.100.12
192.168.100.13
192.168.100.14
[presto@BJ-PRESTO-TEST-100010 ~]$ cat syncPresto.sh
for i in `cat ~/servers.txt`
do
    echo "-----begin to sync Presto file to $i-----"
    scp
/home/presto/lvxin/lvxin-test/presto/presto-server/target/presto-
server-0.107.tar.gz $i:/software/servers/
    ssh $i 'tar zxvf /software/servers/presto-server-0.107.tar.gz -C
/software/servers/'
    echo "-----finish syncing Presto file to $i-----"
done
[presto@BJ-PRESTO-TEST-100010 ~]$ chmod 755 syncPresto.sh
[presto@BJ-PRESTO-TEST-100010 ~]$ ./syncPresto.sh
----- begin to sync Presto file to presto@BJ-PRESTO-TEST-100010.jd.com-----
.
.
```



```

-----finish syncing Presto file to presto@BJ-PRESTO-TEST-100010.jd.com-----
.
.
.
-----finish syncing Presto file to presto@BJ-PRESTO-TEST-100014.jd.com-----

```

2. 修改配置

完成了文件同步之后，需要针对于每个节点上的配置文件进行修改。下面若无特别说明，则需要同时在 Coordinator 和 Worker 上都进行配置项的设置。Presto 集群中的配置文件如下所示。

```

[presto@BJ-PRESTO-TEST-100010 ~]$ ls -l
/software/servers/presto-server-0.107/ etc/
总用量 24
drwxrwxr-x 2 presto presto 4096 7月  8 16:09 catalog
-rwxr-x--- 1 presto presto 2072 7月 13 12:20 config.properties
-rwxr-xr-x 1 presto presto 212 6月 23 15:47 jvm.config
drwxr-x--- 2 presto presto 4096 6月 23 15:43 kafka
-rwxr-xr-x 1 hadp hadp 26 6月 23 15:47 log.properties
-rwxr-xr-x 1 presto presto 124 7月 13 14:22 node.properties
[presto@BJ-PRESTO-TEST-100010 ~]$ ls -l
/software/servers/presto-server-0.107/ etc/kafka
总用量 12
-rw-r----- 1 presto presto 1428 6月 23 15:47 test.topic1.json
[presto@BJ-PRESTO-TEST-100010 ~]$ ls -l
/software/servers/presto-server-0.107/etc/catalog
总用量 36
-rw-r--r-- 1 presto presto 539 6月 23 15:47 hive.properties

```

从上面的输出可以看出 Presto 中的配置文件存在于目录 `/software/servers/presto-server-0.107/etc/` 下，在该目录下存在配置文件 `config.properties`、`jvm.config`、`log.properties`、`node.properties` 和子配置目录 `catalog` 和 `kafka`。下面针对于每个配置文件和子配置目录进行说明。

(1) config.properties

在 Presto 集群中，每个节点上都会启动一个 Presto 服务进程，该配置文件中的配置项会应用于每个 Presto 的服务进程。每个 Presto 服务进程既可以作为 Coordinator，也可以作为 Worker。但是在一个大型的集群中，应该选定一个特定节点上的 Presto 服务进程只作为 Coordinator 提供服务，这样将会提供更加卓越的性能。集群中 Coordinator 上的配置文件 `config.properties` 的内容如下：

```

[presto@BJ-PRESTO-TEST-100010 ~]$ cat config.properties
coordinator=true
node-scheduler.include-coordinator=false
http-server.http.port=8001
task.max-memory=20GB
exchange.min-error-duration=10.00m
exchange.client-threads=64

```

```
task.http-notification-threads=64
query.remote-task.max-consecutive-error-count=20
query.remote-task.min-error-duration=10m
discovery-server.enabled=true
optimizer.optimize-hash-generation=true
query.initial-hash-partitions=26
node-scheduler.multiple-tasks-per-node-enabled=true
distributed-joins-enabled=true
distributed-index-joins-enabled=true
optimizer.optimize-single-distinct=false
discovery.uri=http://BJ-PRESTO-TEST-100010.jd.com:8001
query.queue-config-file=etc/queues.json
```

以集群的模式部署 Presto 服务的时候，配置文件中加黑的配置项，表示在 Coordinator 和 Worker 上的该配置项的值是不同的。对上述配置文件中的配置项说明如下。

coordinator: Presto 集群中的当前节点作为 Coordinator，即当前节点可以接受来自于客户端的查询请求，并且管理查询的执行过程。在 Presto 集群的 Coordinator 节点上的配置文件 \$PRESTO_HOME/etc/config.properties 中，该配置项的值为 true。在 Presto 集群的 Worker 节点上的配置文件 \$PRESTO_HOME/etc/config.properties 中，该配置项的值为 false。需要注意的是，在 Coordinator 的配置文件和 Worker 的配置文件中该配置项的值是不同的。

node-scheduler.include-coordinator: 是否允许在 Coordinator 上执行计算任务。若允许执行计算任务，则 Coordinator 节点除了要接受客户端的查询请求、管理查询的执行过程，还需要执行普通的计算任务。在大型集群中，若在 Coordinator 上执行计算任务，反而会影响查询、生成查询计划和调度的效率，因为对于一些需要大量资源的计算任务（比如全局排序），Coordinator 的大部分资源可能会被用于执行计算任务或者根本不能提供大型计算任务所需要的资源，因此就会对集群中所有查询的调度、管理和监控产生负面影响。

http-server.http.port: 指定 HTTP Server 的端口号。Presto 通过 HTTP 协议进行所有的内部和外部通信。

task.max-memory: 一个单独的计算任务（计算任务是指运行在一个特定的节点上的查询执行计算的一部分，与前面章节所述的 SQLTaskExecution 对应）使用的最大内存数量。内存的大小直接限制了可以进行 Order By 计算的行数。该配置项应该基于查询复杂度、查询的数据量，以及查询并发数进行设置。该配置项设置得太小会限制可以运行的查询，而设置得太高会导致 JVM 的 OOM（Out Of Memory）异常。

discovery-server.enabled: Presto 使用 Discovery 服务来查找集群中的所有节点。每个 Presto 服务在启动的时候都会向 Discovery 服务进行注册。为了简化部署并且避免再多运行一个额外的服务，Presto 的 Coordinator 在启动的时候可以自动运行一个嵌入的 Discovery 服务。该嵌入的 Discovery 服务和 Coordinator 使用同一个 HTTP Server 并使用同一个端口。

discovery.uri: Discovery 服务的 URI。因为我们已经将 Discovery 嵌入了 Coordinator 服务中，因此该 URI 就是 Coordinator 的 URI。例如 URI: http://BJ-PRESTO-TEST-100010.jd.com:8001，该 URI 其实就是 Coordinator 的 URI。需要注意的是，该 URI 不要以 “/” 结尾。

query.queue-config-file: 指定从哪个文件中读取队列配置信息，从上面的配置信息中可

以看出该配置项的值为 `etc/queues.json`。该值指的文件路径为 `$PRESTO_HOME/etc/queues.json`。关于 Presto 中队列的使用，可以参考后面的第 8 章。

此外，若你只有一台服务器用于测试，这台服务器既作为 Coordinator 也作为 Worker，那么就需要将配置属性 `coordinator` 和 `node-scheduler.include-coordinator` 的值均设置为 `true`，如下：

```
[presto@BJ-PRESTO-TEST-single ~]$ cat config.properties
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8001
task.max-memory=20GB
exchange.min-error-duration=10.00m
exchange.client-threads=64
task.http-notification-threads=64
query.remote-task.max-consecutive-error-count=20
query.remote-task.min-error-duration=10m
discovery-server.enabled=true
optimizer.optimize-hash-generation=true
query.initial-hash-partitions=26
node-scheduler.multiple-tasks-per-node-enabled=true
distributed-joins-enabled=true
distributed-index-joins-enabled=true
optimizer.optimize-single-distinct=false
discovery.uri=http://BJ-PRESTO-TEST-single.jd.com:8001
query.queue-config-file=etc/queues.json
```

上述配置内容中加粗显示的配置项表示需要特别注意其配置，因为在分别使用集群模式和单机模式部署 Presto 的时候，这些配置项的值是不同的。

(2) jvm.config

Presto 的开发语言是 Java，每个 Presto 服务进程都是运行在 JVM 之上的，因此需要在 JVM 的配置文件中指定 Presto 服务进程的 Java 运行环境，该配置文件的路径为 `$PRESTO_HOME/etc/jvm.config`。该配置文件中包含了一系列启动 Java 虚拟机时所需要的命令行参数。该配置文件的格式为：每行一个命令行参数。由于该配置文件中的内容不会被 Shell 解析，因此即使某一行的命令行参数包含了空格或者其他特殊的字符，也不需要使引号括起来。例如下面的 `OnOutOfMemoryError` 命令行参数。在 Presto 集群 coordinator 节点上的 JVM 配置文件的内容如下：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat /software/servers/presto-server-0.107/
etc/jvm.config
-server
-Xmx120G
-XX:+UseG1GC
-XX:+ExplicitGCInvokesConcurrent
-XX:+AggressiveOpts
-XX:+HeapDumpOnOutOfMemoryError
-XX:OnOutOfMemoryError=kill -9 %p
-XX:InitiatingHeapOccupancyPercent=75
```



```
-XX:MaxGCPauseMillis=600
```

说明

集群中 Coordinator 和 Worker 上的 JVM 配置文件的内容是一致的。

(3) log.properties

Presto 中日志配置文件为 \$PRSETO_HOME/etc/log.properties, 可以在该配置文件中设置 logger 的最小日志级别。每个 logger 都有一个具有层级关系的名字, 通常 logger 的名字是使用 logger 的类的全名。所有的 logger 都会基于名称拥有一个层级继承关系。在我们部署的集群中日志配置文件的内容为:

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat /software/servers/presto-server-0.107/
etc/log.properties
com.facebook.presto=INFO
```

上面日志级别的设置表示: 对于使用 logger 的类 com.facebook.presto.server 和 com.facebook.presto.hive 来说, 两者的最小日志级别均为 INFO 级别。若不进行配置, 则 Presto 中的 logger 的默认日志级别为 INFO。在 Presto 的日志配置文件中可以设置的日志级别一共有 4 个: INFO、DEBUG、WARN 和 ERROR。

(4) node.properties

在每个 Presto 节点上都需要进行节点属性配置, 节点配置文件的路径是: /software/servers/presto-server-0.107/etc/node.properties。一个 Presto 集群中的节点上的 node.properties 配置文件的内容如下:

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat /software/servers/presto-server-0.107/
etc/node.properties
node.environment=production
node.id=6503c32b-cd96-4eac-87c5-586bbe67450a
node.data-dir=/data1/Logs/presto-server-0.107/data
```

下面对配置文件中的配置项进行说明。

node.environment: Presto 运行环境名称。属于同一个 Presto 集群的 Presto 节点必须拥有相同的环境名称, 即属于同一个 Presto 集群的 Presto 节点的 node.properties 配置文件中的配置属性 node.environment 的值必须相同。

node.id: Presto 集群中的每个 node 的唯一标示。属于同一个 Presto 集群的各个 Presto node 的标示必须是不同的, 可以以 uuid 的值指定该属性的内容, 而 uuid 的值可以通过 Linux 系统命令 uuidgen 进行生成。

node.data-dir: 在每个 Presto Node 所在服务器的操作系统中的路径, Presto 将会在该路径中存放日志和其他的 Presto 数据。

(5) Catalog 配置

Presto 通过 Connector 来访问数据, 一种类型的数据源与一种类型的 Connector 对应。在 Presto 实际使用过程中, 会根据实际的业务需要建立一个或者多个 Catalog, 而每种 Catalog 都有一个特定类型的 Connector 与之对应。因此, 数据源、Connector 和 Catalog 的对应关系

如图 2-1 所示。

从图 2-1 中可以看出数据源与 Connector 是一一对应的，而 Connector 与 Catalog 则是一对多的关系。一个 Catalog 可以含有零个或者多个 Schema，而一个 Schema 中可以含有一个或者多个 Table。例如：一个 Hive 类型 Connector 的 Catalog 就将 Hive 中的 Database 映射成对应的 Schema。所以如果使用 Hive 类型的 Connector 创建了一个与 Hive 数据仓库对应的 Catalog: Catalog1，而 Hive 数据仓库中有一个数据库叫作 Web，而在数据库 Web 中有一张表 Clicks，那么你就可以在 Presto 中需要通过全限定名 catalog1.web.clicks 来访问 Hive 数据仓库中对应的表的数据。

想要在 Presto 中注册一个名为 Catalog1 的新的 Catalog，只需要在 \$PRESTO_HOME/etc/catalog 路径下创建一个名为 catalog1.properties 的 Catalog 的属性配置文件即可。

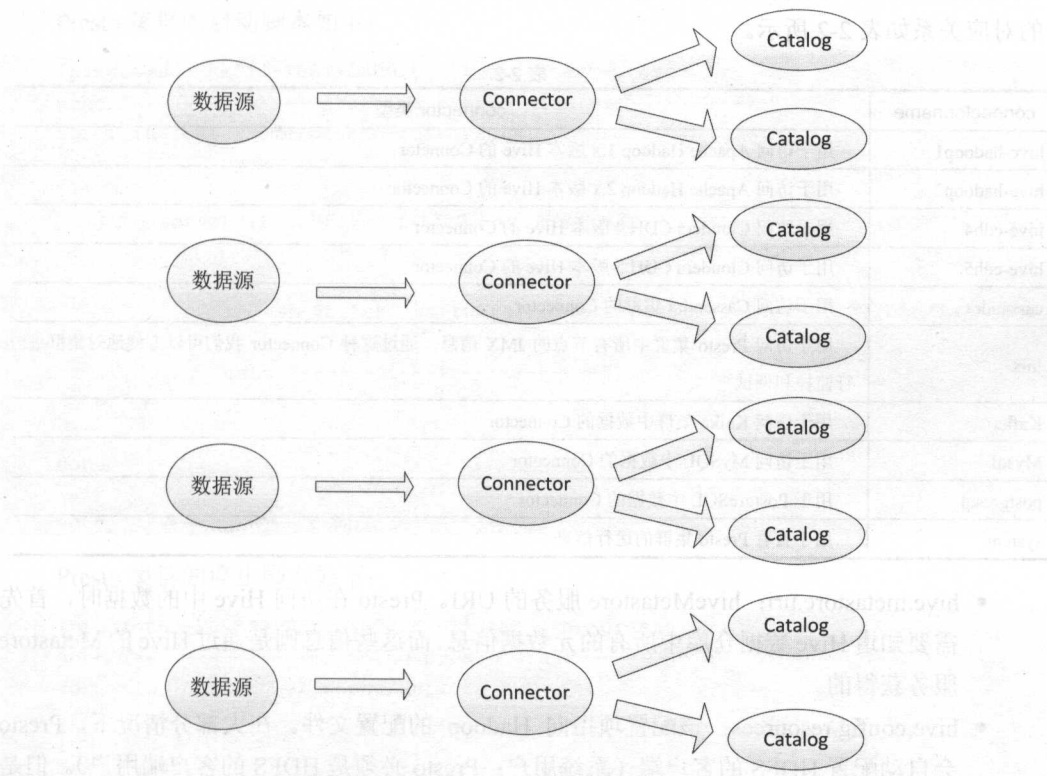


图 2-1

我们以访问 Hive 数据仓库中的数据为例，说明如何创建一个从 Hive 数据仓库中读取数据的 Catalog。由于我们是以 Cluster 的方式部署 Presto 集群的，因此我们只看 Coordinator 上相关的配置文件（若无特别说明，则 Worker 上相应的配置文件与 Coordinator 上的配置文件一致）。

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat /software/servers/presto-server-0.107/etc/catalog/hive.properties
connector.name=hive-cdh5
```

```
hive.metastore.uri=thrift://BJ-PRESTO-TEST-100010.jd.com:9083
hive.config.resources=/software/servers/hadoop-2.5.0-cdh5.3.0/etc/hadoop/core-site.xml,/software/servers/hadoop-2.5.0-cdh5.3.0/etc/hadoop/hdfs-site.xml,/software/servers/hadoop-2.5.0-cdh5.3.0/etc/hadoop/mapred-site.xml
hive.allow-drop-table=true
hive.allow-rename-table=true
[presto@BJ-PRESTO-TEST-100010 ~]$
```

针对该配置文件中的配置项说明如下。

connector.name: Connector 的类型。在上面的配置文件中，该项的值为 hive-cdh5，表示该 Catalog 使用的 Connector 的类型是 hive-cdh5。这种类型的 connector 用于访问 Cloudera CDH5 版本的 Hive。connector.name 的不同值对应不同类型的 Connector。对于其他的 Connector 类型，将会在核心设计篇进行详细讲解。connector.name 的取值与 connector 类型的对应关系如表 2-2 所示。

表 2-2

connector.name	connector 类型
hive-hadoop1	用于访问 Apache Hadoop 1.x 版本 Hive 的 Connector
hive-hadoop2	用于访问 Apache Hadoop 2.x 版本 Hive 的 Connector
hive-cdh4	用于访问 Cloudera CDH 4 版本 Hive 的 Connector
hive-cdh5	用于访问 Cloudera CDH 5 版本 Hive 的 Connector
cassandra	用于访问 Cassandra 集群的 Connector
jmx	用于访问 Presto 集群中所有节点的 JMX 信息。通过这种 Connector 我们可以方便地对集群进行监控和调试
Kafka	用于访问 Kafka 集群中数据的 Connector
Mysql	用于访问 MySQL 中数据的 Connector
postgresql	用于 PostgreSQL 中数据的 Connector
system	用于查看 Presto 集群的运行信息

- **hive.metastore.uri:** hiveMetastore 服务的 URI。Presto 在访问 Hive 中的数据时，首先需要知道 Hive 数据仓库中所有的元数据信息，而这些信息则是通过 Hive 的 Metastore 服务获得的。
- **hive.config.resources:** 该配置项指向 Hadoop 的配置文件。在大部分情况下，Presto 会自动配置 HDFS 的客户端（系统用户：Presto 必须是 HDFS 的客户端用户）。但是在有些情况下（例如使用 federated HDFS 的时候），需要指定一些额外的 HDFS 客户端选项，才能访问 HDFS 集群，因此需要在以 Hive 类型数据源的 Catalog 的配置文件中添加属性 hive.config.resources，使其指向 Hadoop 的配置文件。
- **hive.allow-drop-table:** 是否允许在 Presto 中删除 Hive 中的表。
- **hive.allow-rename-table:** 是否允许在 Presto 中重命名 Hive 中的表。

(6) Kafka 配置

在读取 Kafka 中的数据的时候，需要使用 kafka-connector。由于 Kafka 中的数据是非结构化的，因此需要将 Kafka 中的每条日志都映射为表中的一行，这种 Kafka 中的日志与表结

构的映射关系是通过名为“表描述文件”的配置文件来完成的。在默认情况下，这些表描述文件位于 \$PRESTO_HOME/etc/kafka/ 目录下，不过也可以通过配置项 `kafka.table-description-dir` 来更改表描述文件的路径。关于 Kafka 以及其表描述文件的详细配置和说明，可以查看第 11 章。

2. 启动和停止服务

按照前面的步骤完成了 Presto 集群的安装部署之后，就可以对整个 Presto 集群进行服务启动了。目前 Presto 没有提供统一的集群启动和停止脚本，需要自己到每台节点上进行 Presto 服务的启动和停止。

但是由于在前面准备工作中建立了 Coordinator 到各个 Worker 的 SSH 信任关系。因此可以通过 Shell 脚本完成整个集群所有节点上 Presto 服务的启动和停止。

Presto 集群的启动脚本如下：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat start-presto.sh
echo "-----starting presto cluster-----"
for i in `cat /home/presto/servers.txt`
do
{
    #若在 servers.txt 中的某一行以“#”开头，则忽略此行
    if [[ ${i:0:1} != "#" ]]
    then
        r=`ssh $i "sh /software/servers/presto-server-0.107/bin/launcher
start"`
        echo "-----$i---$r-----"
    fi
} &
done
wait
echo "-----complete-----"
```

Presto 集群的停止脚本如下：

```
[presto@BJ-PRESTO-TEST-100010 ~]$ cat stop-presto.sh
echo "-----stoping presto start-----"
for i in `cat /home/presto/servers.txt`
do
{
    #若在 servers.txt 中的某一行以“#”开头，则忽略此行
    if [[ ${i:0:1} != "#" ]]
    then
        r=`ssh $i "sh /software/servers/presto-server-0.107/bin/launcher
stop"`
        echo "-----$i---$r-----"
    fi
} &
done
wait
echo "-----complete-----"
```

```
[presto@BJ-PRESTO-TEST-100010 ~]$
```

2.4.2 客户端部署

上面讲述了如何完成 Presto 集群服务的安装、部署和启动，之后就可以通过 Presto 的客户端访问集群并执行数据查询和计算了。本节主要说明如何进行 Presto 客户端的部署和使用。

1. 部署

前面已经说明，Presto 的客户端是由子工程 `presto-cli` 源码编译得到的，其编译后的可执行文件为 `presto-cli-0.107-executable.jar`。因此需要使用该 `jar` 文件进行客户端部署，部署步骤如下。

```
[presto@BJ-PRESTO-TEST-100010 ~]$ chmod 755 presto-cli-0.107-executable.jar
[presto@BJ-PRESTO-TEST-100010 ~]$ ./presto-cli-0.107-executable.jar --help
NAME
    presto - Presto interactive console

SYNOPSIS
    presto [--catalog <catalog>] [--debug] [--execute <execute>]
    [(-f <file> | --file <file>)] [(-h | --help)]
    [--output-format <output-format>] [--schema <schema>]
    [--server <server>] [--session <session>...]
    [--socks-proxy <socks-proxy>] [--source <source>] [--user <user>]
    [--version]

OPTIONS
    --catalog <catalog>
        Default catalog

    --debug
        Enable debug information

    --execute <execute>
        Execute specified statements and exit

    -f <file>, --file <file>
        Execute statements from file and exit

    -h, --help
        Display help information

    --output-format <output-format>
        Output format for batch mode (default: CSV)

    --schema <schema>
        Default schema

    --server <server>
        Presto server location (default: localhost:8080)
```



```

--session <session>
    Session property (property can be used multiple times; format is
    key=value)

--socks-proxy <socks-proxy>
    SOCKS proxy to use for server connections

--source <source>
    Name of source making query

--user <user>
    Username

--version
    Display version information and exit

```

通过上面的命令可以看出，通过--help 可以查看 presto-cli 客户端的使用说明。这里对使用说明中的每一项就不再进行赘述了。

2. 启动

若针对于已经部署完成的 Presto 客户端，则其启动命令如下：

```

presto-cli-0.107-executable.jar --server 192.168.100.10:8001 --catalog hive
--schema employees
presto:employees> show tables;
    Table
    -----
employees
(22 rows)
Query 20150910_021639_00002_zgnuq, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:00 [22 rows, 574B] [556 rows/s, 14.2KB/s]

presto:employees> select count(*) from employees;
_col0
-----
300024
(1 row)
Query 20150910_021657_00003_zgnuq, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:01 [300K rows, 7.48MB] [262K rows/s, 6.54MB/s]
presto:employees> select * from employees limit 1;
emp_no | birth_date | first_name | last_name | gender | hire_date
-----+-----+-----+-----+-----+-----
10001 | 1953-09-03 | Georgi | Facello | M | 1986-06-26
(1 row)
Query 20150910_021718_00004_zgnuq, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:00 [5.38K rows, 4.32MB] [36.1K rows/s, 29MB/s]

```

下面对上面的客户端启动命令进行说明。

--server: 指定 Presto 集群中 Coordinator 的 URI, 其内容为 IP 地址:端口号。

--catalog: 指定 Presto-Cli 客户端启动之后使用的默认 Catalog。

--schema: 指定 Presto-Cli 客户端启动之后使用的默认 Schema。

Presto-cli 客户端一旦启动完成就可以使用 ANSI 标准 SQL 进行数据查询和计算了。

2.4.3 JDBC 使用

Presto 提供了通过 JDBC 访问集群的方式, 使用者可以在 Java 程序中通过 Presto 的 JDBC 驱动连接到 Presto 集群, 从而进行数据查询和分析。本节会详细介绍在 Eclipse 中如何开发一个简单的 Java 程序, 使用 JDBC 驱动访问 Presto 集群。

(1) 新建一个 Java Maven 工程: PrestoJDBCClient。

(2) 修改该 Maven 工程的 pom.xml 文件, 添加 Presto-0.107 的 JDBC 驱动, pom.xml 文件的内容如下:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.jd</groupId>
    <artifactId>PrestoJDBCClient</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>PrestoJDBCClient</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.facebook.presto</groupId>
            <artifactId>presto-jdbc</artifactId>
            <version>0.107</version>
        </dependency>
    </dependencies>
</project>
```

(3) 开发客户端代码, 在包 com.jd.PrestoJDBCClient 下创建一个类 PrestoJDBCTestClient:

```

package com.jd.PrestoJDBCClient;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.TimeZone;

import com.facebook.presto.jdbc.PrestoConnection;
import com.facebook.presto.jdbc.PrestoStatement;

public class PrestoJDBCTestClient {

    public static void printRow(ResultSet rs, int[] types) throws SQLException {
        for (int i = 0; i < types.length; i++)
        {
            System.out.print(" ");
            System.out.print(rs.getObject(i + 1));
        }
        System.out.println("");
    }

    public static void connect() throws SQLException {
        // 设置时区, 这里必须要设置
        TimeZone.setDefault(TimeZone.getTimeZone("Asia/Shanghai"));
        try {
            //加载 Presto JDBC 驱动类
            Class.forName("com.facebook.presto.jdbc.PrestoDriver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        PrestoConnection connection = null;
        try {
            // Presto 连接串, 在连接串中指定了默认的 catalog 为: system, 默认的 schema 为:
            runtime, 使用的用户名为: guest1, 这个用户名根据实际业务自己设定, 用来标示执行 SQL 的用户, 虽然不会
            通过该用户名进行身份认证, 但是必须要写。密码直接指定为 null, 或者可以随便指定一个任意密码, Presto
            是不会对密码进行验证的
            connection = (PrestoConnection) DriverManager.getConnection(
                "jdbc:presto://192.168.100.10:8001/system/runtime",
                "guest1", null);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        PrestoStatement statement = null;
        try {
            statement = (PrestoStatement) connection.createStatement();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        String query = "select * from nodes";
        ResultSet rs = null;
        try {
            rs = statement.executeQuery(query);
        } catch (SQLException e) {

```



```

        e.printStackTrace();
    }

    int cn = rs.getMetaData().getColumnCount();
    int[] types = new int[cn];
    for (int i = 1; i <= cn; i++) {
        types[i - 1] = rs.getMetaData().getColumnType(i);
    }

    try {
        while (rs.next()) {
            printRow(rs, types);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws ClassNotFoundException,
    SQLException {
    connect();
}
}

```

(4) 运行。

选中类文件，单击鼠标右键，再选择“Run as”→“Java Application”命令。运行上面开发的 Java 程序，可以得到如下输出结果：

```

6503c32b-cd96-4eac-87c5-586bbe67450a http://192.168.100.10:8001 presto-main:
0.100 true
5378d8b3-b604-45ca-8bf4-f699d0796551 http://192.168.100.11:8001 presto-main:
0.100 true
ad5f4277-0316-43c6-b19f-0dc6eefa25b7 http://192.168.100.12:8001 presto-main:
0.100 true
cf2b3285-a03f-4ed7-957f-2c776b105b81 http://192.168.100.13:8001
presto-main:0.100 true
b15b7765-be43-4204-ab94-78a766172208 http://192.168.100.14:8001 presto-main:
0.100 true

```

至此，已经完成使用 Presto JDBC 驱动进行数据查询和计算的 Java 程序的开发。

2.5 小结

本章深入、细致地介绍了 Presto 安装部署的环境说明、准备工作、源码编译、安装部署和集群使用内容，通过本章的学习，读者可以轻松地完成 Presto 集群的安装部署，并且可以参照本章中的代码直接使用 presto-jdbc 驱动完成 Java 应用程序的开发。

第二部分

核心设计篇

查询的提交、Presto 集群中任务的调度、节点的管理、查询信息的更新和数据的传输，都依赖于 Presto 集群中构建的 RESTful 服务。在这一部分，我们会对 Presto 提供的 RESTful 服务以及查询执行过程进行详细说明，并对关键步骤进行源码解析，通过这一部分，读者可以对 Presto 的整个查询执行过程拥有全面深入的理解。

第 3 章

Presto RESTful 框架解析

在 Presto 中，几乎所有的操作都需要依赖于基于 AirLift 框架构建的 RESTful 服务来完成，包括 Worker 节点的管理、查询语句的提交、查询执行状态的显示、各个 Task 之间数据的传递等。因此 Presto 中的 RESTful 服务是 Presto 集群的基础。

本章假设读者已经熟悉 RESTful 服务的概念、基础架构和作用。不会再对如何搭建 RESTful 服务进行讲解，主要对 Presto 提供的所有 RESTful 服务接口进行详细说明，从而使读者对 Presto 提供的所有 RESTful 接口拥有一个全面的了解。

Presto 总共提供了 4 类 RESTful 接口，分别是 Statement 接口、Query 接口、Stage 接口和 Task 接口。下面对这 4 种 RESTful 接口进行详细说明。

3.1 Statement 服务接口

与 SQL 语句相关的请求均由该服务接口处理，包括接受提交的 SQL 语句、获取查询执行的结果、取消查询等。Statement 服务接口的实现类为 StatementResource，该类的源码如下：

```
//从这里可以看出，所有路径为/v1/statement 的 URL 请求都交给该类提供的 RESTful 服务处理
@Path("/v1/statement")
public class StatementResource {
    //地址为:/v1/statement 的 POST 请求由以下方法处理
    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response createQuery(
        String statement,
        @Context HttpServletRequest servletRequest,
        @Context UriInfo uriInfo) throws InterruptedException {
        assertRequest(!isEmpty(statement), "SQL statement is empty");
        Session session = createSessionForRequest(servletRequest);
        ExchangeClient exchangeClient = exchangeClientSupplier.get();
        //在该方法中执行查询
    }
}
```

```

        Query query = new Query(session, statement, queryManager, exchangeClient);
        queries.put(query.getId(), query);
        //获得查询的结果并组装成 Response 进行返回, 从该方法中可以看出第二个参数是空, 若该参数
        数为空, 则分批返回结果
        return getQueryResults(query, Optional.empty(), uriInfo, new Duration(1,
MILLISECONDS));
    }
    .....
    private static Response getQueryResults(Query query, Optional<Long> token,
UriInfo uriInfo, Duration wait) throws InterruptedException {
        QueryResults queryResults;
        //若 token 不为空, 则执行 query.getResults 方法, 否则就执行 query.getNextResults
        方法, 因为在 createQuery 方法中传入的该参数为: Optional.empty(), 因此其值为空, 则执行方法:
        query.getNextResults 方法。注意: Query 实际上是 StatementResource 的内部类
        if (token.isPresent()) {
            queryResults = query.getResults(token.get(), uriInfo, wait);
        }
        else {
            queryResults = query.getNextResults(uriInfo, wait);
        }
        .....
        //将查询结果封装进 Response 中
        ResponseBuilder response = Response.ok(queryResults);
        .....
        return response.build();
    }

```

//地址为: /v1/statemen/queryID/token 的 GET 请求由以下方法处理, 该方法根据 Token 值返回 queryID 对应的查询的部分执行结果, 这里的 Token 主要用于保证分批读取查询结果的顺序。前面章节所说的客户端会不断向 Coordinator 获取部分的查询执行结果, 直到获得了所有的结果。在这个过程中, 每次获得部分查询执行结果的请求都是由该方法进行处理的。

```

@GET
@Path("/{queryId}/{token}")
@Produces(MediaType.APPLICATION_JSON)
public Response getQueryResults(
    @PathParam("queryId") QueryId queryId,
    @PathParam("token") long token,
    @QueryParam("maxWait") Duration maxWait,
    @Context UriInfo uriInfo)
    throws InterruptedException {
    Query query = queries.get(queryId);
    if (query == null) {
        return Response.status(Status.NOT_FOUND).build();
    }

    Duration wait = WAIT_ORDERING.min(MAX_WAIT_TIME, maxWait);
    return getQueryResults(query, Optional.of(token), uriInfo, wait);
}

```

//地址为: /v1/statemen/queryID/token 的 DELETE 请求, 由以下方法处理, 该方法用于取消一个 Query。在该方法中并没有使用 Token, 因此 Token 在这里的作用只是用于区别于 QueryResource 中的 cancelQuery 方法, 而且该方法与 QueryResource 中的 cancelQuery 方法的区别还在于加入了一些特殊情况

况处理和 exchangeClient 的关闭与清理。

```

@DELETE
@Path("/{queryId}/{token}")
@Produces(MediaType.APPLICATION_JSON)
public Response cancelQuery(@PathParam("queryId") QueryId queryId,
    @PathParam("token") long token)
{
    Query query = queries.get(queryId);
    if (query == null) {
        return Response.status(Status.NOT_FOUND).build();
    }
    query.close();
    return Response.noContent().build();
}

@ThreadSafe
// Query 是 StatementResource 的内部类
public static class Query{
    .....
    public synchronized QueryResults getNextResults(UriInfo uriInfo, Duration
maxWaitTime) throws InterruptedException {
        //获得查询执行结果，只等待 maxWaitTime 指定的时间，若该时间内计算还没有处理完毕，就只获得该
时间内处理完毕的结果数据
        Iterable<List<Object>> data = getData(maxWaitTime);
        .....//查询后续处理：连接关闭、状态更新等工作

        //若查询还没有结束或者还有更多的数据需要发送，则构建一个 nextURI 用于下一次请求剩余
的数据。
        URI nextResultsUri = null;
        if ((!queryInfo.getState().isDone()) || (!exchangeClient.isClosed())) {
            nextResultsUri = createNextResultsUri(uriInfo);
        }
        //从这里可以看到 nextResultUri 被作为 Result 的一部分返回给了 Client 端，Client
端会使用 nextResultUri 来请求后续的查询处理结果
        QueryResults queryResults = new QueryResults(queryId.toString(),
uriInfo.getRequestUriBuilder().replaceQuery("").replacePath(queryInfo.getSelf().ge
tPath()).build(),
            findCancelableLeafStage(queryInfo),
            nextResultsUri,
            columns,
            data,
            toStatementStats(queryInfo),
            toQueryError(queryInfo),
            queryInfo.getUpdateType(),
            updateCount);
        .....//后续处理
        return queryResults;
    }

    private synchronized URI createNextResultsUri(UriInfo uriInfo) {
        //构建形如： /v1/statement/queryID/token 的 RestFul 请求
        return
uriInfo.getBaseUriBuilder().replacePath("/v1/statement").path(queryId.toString()).
path(String.valueOf(resultId.incrementAndGet())).replaceQuery("").build();
    }
}

```



```

    }
    public synchronized QueryResults getResults(long token, UriInfo uriInfo, Duration
maxWaitTime) throws InterruptedException {
        //如果这次请求结果的 URI 与上次请求结果的 URI 相同, 则直接返回上次的结果
        String requestedPath = uriInfo.getAbsolutePath().getPath();
        if (lastResultPath != null && requestedPath.equals(lastResultPath)) {
            queryManager.getQueryInfo(queryId);
            queryManager.recordHeartbeat(queryId);
            return lastResult;
        }
        .....//异常情况处理
        //返回 next Results
        return getNextResults(uriInfo, maxWaitTime);
    }
}
}

```

3.2 Query 服务接口

与查询执行相关的 RESTful 请求均由 Query 服务接口处理, 包括 SQL 语句的提交、获取查询执行的结果、取消查询等。Query 服务接口的实现类为 QueryResource, 该类的源码如下:

```

//所有路径为/v1/query 的 URL 请求都交给该类提供的 RESTful 服务处理
@Path("/v1/query")
public class QueryResource {
    //地址为: /v1/query/queryID 的 GET 请求由以下方法处理, 该方法返回一个查询执行的状态信息
    @GET
    @Path("/{queryId}")
    public Response getQueryInfo(@PathParam("queryId") QueryId queryId)
    {
        checkNotNull(queryId, "queryId is null");

        try {
            QueryInfo queryInfo = queryManager.getQueryInfo(queryId);
            return Response.ok(queryInfo).build();
        }
        catch (NoSuchElementException e) {
            return Response.status(Status.GONE).build();
        }
    }
    //地址为: /v1/query 的 POST 请求由以下方法处理, 该方法生成一个新的查询
    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response createQuery(
        String statement,
        @Context HttpServletRequest servletRequest,
        @Context UriInfo uriInfo) {
        assertRequest(!isEmpty(statement), "SQL statement is empty");
    }
}

```

```

        Session session = createSessionForRequest(servletRequest);

        QueryInfo queryInfo = queryManager.createQuery(session, statement);
        URI pagesUri =
uriBuilderFrom(uriInfo.getRequestUri()).appendPath(queryInfo.getQueryId().toString
()).build();
        return Response.created(pagesUri).entity(queryInfo).build();
    }

    //地址为: /v1/query/queryID 的 DELETE 请求由以下方法处理, 该方法取消一个查询
    @DELETE
    @Path("/{queryId}")
    public void cancelQuery(@PathParam("queryId") QueryId queryId) {
        checkNotNull(queryId, "queryId is null");
        queryManager.cancelQuery(queryId);
    }

    //地址为: /v1/query/stage/stageID 的 DELETE 请求由以下方法处理, 该方法取消一个 Stage
    @DELETE
    @Path("/stage/{stageId}")
    public void cancelStage(@PathParam("stageId") StageId stageId)
    {
        checkNotNull(stageId, "stageId is null");
        queryManager.cancelStage(stageId);
    }
}

```

3.3 Stage 服务接口

与 Stage 相关的 RESTful 请求均由 Stage 服务接口处理, 其实该接口只提供了一个功能, 就是取消或者结束一个指定的 Stage。Stage 服务接口的实现类为 StageResource, 该类的源码如下:

```

//所有路径为/v1/stage 的 URL 请求都交给该类提供的 RestFul 服务处理
@Path("/v1/stage")
public class StageResource {
    //地址为: /v1/stage/stageID 的 DELETE 请求由以下方法处理, 该方法取消或者提前结束一个 Stage
    @DELETE
    @Path("/{stageId}")
    public void cancelStage(@PathParam("stageId") StageId stageId)
    {
        checkNotNull(stageId, "stageId is null");
        queryManager.cancelStage(stageId);
    }
}

```

3.4 Task 服务接口

与 Task 相关的 RESTful 请求均由 Task 服务接口处理，包括 Task 的创建、更新、状态查询和结果查询等。Task 服务接口的实现类为 TaskResource，该类的源码如下：

```
//所有路径为/v1/task 的 URL 请求都交给该类提供的 RestFul 服务处理
@Path("/v1/task")
public class TaskResource {
    //URI 为: /v1/task/taskID 的 Post 均由以下方法处理，这种请求主要包括两种：创建一个新的 Task
    //或者更新 Task 的状态
    @POST
    @Path("/{taskId}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response createOrUpdateTask(@PathParam("taskId") TaskId taskId,
    TaskUpdateRequest taskUpdateRequest, @Context UriInfo uriInfo)
    {
        checkNotNull(taskUpdateRequest, "taskUpdateRequest is null");
        //若存在 taskId 对应的 Task，就根据 taskUpdateRequest 中的内容更新该 Task；否则就
        //根据 taskUpdateRequest 中的内容创建一个新的 Task
        TaskInfo taskInfo =
        taskManager.updateTask(taskUpdateRequest.getSession(),
            taskId,
            taskUpdateRequest.getFragment(),
            taskUpdateRequest.getSources(),
            taskUpdateRequest.getOutputIds());

        if (shouldSummarize(uriInfo)) {
            taskInfo = taskInfo.summarize();
        }
        return Response.ok().entity(taskInfo).build();
    }

    //URI 为: /v1/task/taskId 的 Get 请求，均由以下方法处理，下面方法的作用是根据 taskId 获得指
    //定的 Task 的相关信息，并返回给客户端
    @GET
    @Path("/{taskId}")
    @Produces(MediaType.APPLICATION_JSON)
    public void getTaskInfo(@PathParam("taskId") final TaskId taskId,
        @HeaderParam(PRESTO_CURRENT_STATE) TaskState currentState,
        @HeaderParam(PRESTO_MAX_WAIT) Duration maxWait,
        @Context UriInfo uriInfo,
        @Suspended AsyncResponse asyncResponse)
    {
        checkNotNull(taskId, "taskId is null");

        if (currentState == null || maxWait == null) {
            TaskInfo taskInfo = taskManager.getTaskInfo(taskId);
            if (shouldSummarize(uriInfo)) {
                taskInfo = taskInfo.summarize();
            }
            asyncResponse.resume(taskInfo);
        }
    }
}
```



```

        return;
    }

    ListenableFuture<TaskInfo> futureTaskInfo = MoreFutures.addTimeout(
        taskManager.getTaskInfo(taskId, currentState),
        () -> taskManager.getTaskInfo(taskId),
        maxWait,
        executor);

    if (shouldSummarize(uriInfo)) {
        futureTaskInfo = Futures.transform(futureTaskInfo,
TaskInfo::summarize);
    }

    // For hard timeout, add an additional 5 seconds to max wait for thread
scheduling contention and GC
    Duration timeout = new Duration(maxWait.toMillis() + 5000, MILLISECONDS);
    bindAsyncResponse(asyncResponse, futureTaskInfo, executor)
        .withTimeout(timeout);
}

//URI 为: /v1/task/taskId 的 delete 请求, 均由以下方法处理, 以下方法将 TaskId 对应的 Task
进行删除或者提前结束
@DELETE
@Path("/{taskId}")
@Produces(MediaType.APPLICATION_JSON)
public Response deleteTask(@PathParam("taskId") TaskId taskId,
    @QueryParam("abort") @DefaultValue("true") boolean abort,
    @Context UriInfo uriInfo)
{
    checkNotNull(taskId, "taskId is null");

    TaskInfo taskInfo;
    if (abort) {
        taskInfo = taskManager.abortTask(taskId);
    }
    else {
        taskInfo = taskManager.cancelTask(taskId);
    }

    if (shouldSummarize(uriInfo)) {
        taskInfo = taskInfo.summarize();
    }
    return Response.ok(taskInfo).build();
}

//URI 为: /v1/task/taskId/results/outputId/token 的 Get 请求均由以下方法处理, 该方法用
于获得 TaskId 指定的 Task 生成的用于输出给某个下游 Task (该 Task 由 outputId 标示) 的数据
@GET
@Path("/{taskId}/results/{outputId}/{token}")
@Produces(PRESTO_PAGES)
public void getResults(@PathParam("taskId") TaskId taskId,
    @PathParam("outputId") TaskId outputId,
    @PathParam("token") final long token,

```



```

        @Suspended AsyncResponse asyncResponse)
        throws InterruptedException
    {
        checkNotNull(taskId, "taskId is null");
        checkNotNull(outputId, "outputId is null");

        ListenableFuture<BufferResult> bufferResultFuture =
taskManager.getTaskResults(taskId, outputId, token, DEFAULT_MAX_SIZE);
        bufferResultFuture = MoreFutures.addTimeout(
            bufferResultFuture,
            () -> BufferResult.emptyResults(token, false),
            DEFAULT_MAX_WAIT_TIME,
            executor);

        ListenableFuture<Response> responseFuture =
Futures.transform(bufferResultFuture, (BufferResult result) -> {
            List<Page> pages = result.getPAGES();

            GenericEntity<?> entity = null;
            Status status;
            if (!pages.isEmpty()) {
                entity = new GenericEntity<>(pages, new TypeToken<List<Page>>()
{}.getType());
                status = Status.OK;
            }
            else if (result.isBufferClosed()) {
                status = Status.GONE;
            }
            else {
                status = Status.NO_CONTENT;
            }

            return Response.status(status)
                .entity(entity)
                .header(PRESTO_PAGE_TOKEN, result.getToken())
                .header(PRESTO_PAGE_NEXT_TOKEN, result.getNextToken())
                .build();
        });

        // For hard timeout, add an additional 5 seconds to max wait for thread
        scheduling contention and GC
        Duration timeout = new Duration(DEFAULT_MAX_WAIT_TIME.toMillis() + 5000,
MILLISECONDS);
        bindAsyncResponse(asyncResponse, responseFuture, executor)
            .withTimeout(timeout,
                Response.status(Status.NO_CONTENT)
                    .header(PRESTO_PAGE_TOKEN, token)
                    .header(PRESTO_PAGE_NEXT_TOKEN, token)
                    .build());
    }

    //URI 为: /v1/task/taskId/results/outputId 的 Delete 请求均由以下方法处理, 该方法用于清
    除 TaskId 标示的 Task 生成的, 用于输出给下游 Task (该 Task 由 outputId 标示) 的数据

```

```
@DELETE
@Path("/{taskId}/results/{outputId}")
@Produces(MediaType.APPLICATION_JSON)
public Response abortResults(@PathParam("taskId") TaskId taskId,
@PathParam("outputId") TaskId outputId, @Context UriInfo uriInfo)
{
    checkNotNull(taskId, "taskId is null");
    checkNotNull(outputId, "outputId is null");

    TaskInfo taskInfo = taskManager.abortTaskResults(taskId, outputId);
    if (shouldSummarize(uriInfo)) {
        taskInfo = taskInfo.summarize();
    }
    return Response.ok(taskInfo).build();
}
```

3.5 小结

Presto 集群中的数据传输、节点通信、心跳感应、计算监控、计算调度和计算分布全部都是基于 RESTful 服务实现的，因此 Presto 中的 RESTful 服务就是 Presto 所有服务的基石。本章针对 Presto 中的 Statement、Query、Stage 和 Task 类型的 RESTful 接口进行了说明，并通过源码对这些接口进行了详细的分析，从而使读者对 Presto 提供的 RESTful 接口深入了解，并为后续章节的阅读打下坚实的基础。

第4章

提交查询

4.1 提交查询的步骤

终端用户可以通过 JDBC 或者 CLI 提交查询语句（当然也可以使用官方资源地址 <https://presto.io/resources.html> 上提供的由第三方机构或者个人使用 Python、C 等语言开发的驱动提交查询），本章会以 CLI 为例来讲解 Presto 如何提交查询。

Presto 客户端对查询语句的提交主要分为以下 3 个步骤。

- (1) 从指定的文件、命令行参数或者 Cli 窗口中获取需要执行的 SQL 语句。
- (2) 将得到的 SQL 语句组装成一个 RESTful 请求，发送给 Coordinator，并处理返回的 response。
- (3) Cli 会不停地循环分批读取查询结果并在屏幕进行动态显示，直到查询结果完全显示完毕。

4.2 源码解析

本节会从源码级别对 CLI 提交查询的过程进行详细分析。

Presto 的 Cli 客户端启动类为 `com.facebook.presto.cli.Presto`，该类的源码如下：

```
public final class Presto
{
    private Presto() {}
    public static void main(String[] args) throws Exception {
        //根据传递的参数初始化一个 Console 对象，该对象中保存了启动 Cli 时传入的所有参数
        Console console = singleCommand(Console.class).parse(args);
        //若启动的时候使用了--help 或者--version 则会显示帮助信息或者版本信息，然后直接退出
        if (console.helpOption.showHelpIfRequested() ||
            console.versionOption.showVersionIfRequested()) {
            return;
        }
    }
}
```



```

//进入主程序，然后根据启动 CLI 传入的不同参数进行不同的处理
console.run();
}
}

```

从上面的代码可以看出，如果启动 Presto CLI 的时候没有使用 `--help` 或 `--version` 参数，则运行 `console.run()` 方法对 SQL 语句进行处理，`com.facebook.presto.cli.Console` 类的关键代码如下：

```

@Override
public void run()
{
    //获取所有的 client Session 参数，这些参数被组装到 ClientSession 类中
    ClientSession session = clientOptions.toClientSession();
    //如果 Cli 启动的时候指定了--execute 参数，则 hasQuery 的值为 true，说明需要直接
    执行通过 shell 传入的 SQL 语句
    boolean hasQuery = !Strings.isNullOrEmpty(clientOptions.execute);
    //如果 Cli 启动的时候指定了-f 或者--file 参数，则 isFromFile 的值为 true，说明读
    取文件中的 SQL 语句执行
    boolean isFromFile = !Strings.isNullOrEmpty(clientOptions.file);

    if (!hasQuery || !isFromFile) {
        AnsiConsole.systemInstall();
    }
    //初始化日志
    initializeLogging();
    //获取--execute 参数对应的内容
    String query = clientOptions.execute;
    if (hasQuery) {
        query += ";";
    }

    if (isFromFile) {
        if (hasQuery) {
            //在启动 Cli 的时候不能同时指定--execute 和--file 参数
            throw new RuntimeException("both --execute and --file specified");
        }
        try {
            //从文件中读取 SQL 语句
            query = Files.toString(new File(clientOptions.file), UTF_8);
            //启动 Cli 的时候即使没有使用--execute 参数，但是指定了--file 参数，也可以将
            hasQuery 标示属性设置为 true，此时该标示属性的意义已经改变，不再标示是否指定
            了--execute 属性，而是标示是否在启动 CLI 的时候就指定了 SQL 语句。
            hasQuery = true;
        }
        catch (IOException e) {
            throw new RuntimeException(format("Error reading from file %s: %s",
clientOptions.file, e.getMessage()));
        }
    }
    // 生成一个查询执行包装类，后续的查询都会通过该类启动和执行。
    try (QueryRunner queryRunner = QueryRunner.create(session,

```



```
Optional.ofNullable(clientOptions.socksProxy))) {
    //若启动 Cli 的时候指定了--execute 或者--file 参数, 则执行方法:
    executeCommand(...), 否则就直接启动 Cli, 接受终端用户在 Cli 中的输入, 并提交查询。
    if (hasQuery) {
        //直接提交 SQL 语句
        executeCommand(queryRunner, query, clientOptions.outputFormat);
    }
    else {
        //启动 Cli 窗口, 接受用户输入, 分析 SQL 语句并提交。
        runConsole(queryRunner, session);
    }
}
}
```

通过上面的代码我们可以看到, 根据在启动 Cli 的时候是否指定 `execute` 或者 `file` 参数, Presto Cli 对查询语句有两种不同的处理方式: 直接提交 SQL 语句或者启动 Cli 终端接受并解析用户的输入, 然后提交语句。

本节的目的是让读者深入了解查询提交的步骤, 所以这里不关注 Cli 如何解析终端用户输入。下面的内容主要分析直接提交带有 `execute` 参数时, SQL 语句提交和执行的流程。`executeCommand(queryRunner, query, clientOptions.outputFormat)`方法也是位于类 `com.facebook.presto.cli.Console` 中的, 该方法的主要作用就是对输入的 SQL 语句进行解析并执行, 其代码如下:

```
private static void executeCommand(QueryRunner queryRunner, String query,
OutputFormat outputFormat) {
    // StatementSplitter 对输入的连续 SQL 语句进行切分, 默认是为字符“;”作为 SQL 语
    句的分隔符
    StatementSplitter splitter = new StatementSplitter(query);
    for (Statement split : splitter.getCompleteStatements()) {
        if (!isEmptyStatement(split.statement())) {
            //获取一条 SQL 语句并进行处理
            process(queryRunner, split.statement(), outputFormat, false);
        }
    }
    if (!isEmptyStatement(splitter.getPartialStatement())) {
        System.err.println("Non-terminated statement: " + splitter.
            getPartialStatement());
    }
}
//该方法由上面的代码调用, 因此传入的 interactive 参数的值为 false
private static void process(QueryRunner queryRunner, String sql, OutputFormat
outputFormat, boolean interactive) {
    //执行查询
    try (Query query = queryRunner.startQuery(sql)) {
        //输出结果, 请注意: 上面的执行查询只是发起一次 RESTful 请求而已, 而前面所述的 Cli
        循环分批显示查询结果, 则是在该方法中循环发送不同的 RESTful 请求来实现的。
        query.renderOutput(System.out, outputFormat, interactive);
        if (!query.getSetSessionProperties().isEmpty() || !query.
            getResetSessionProperties().isEmpty()) {
            Map<String, String> sessionProperties = new HashMap<>(queryRunner.
```

```

        getSession().getProperties());
        sessionProperties.putAll(query.getSetSessionProperties());
        sessionProperties.keySet().removeAll(query.
            getResetSessionProperties());
        queryRunner.setSession(withProperties(queryRunner.getSession(),
            sessionProperties));
    }
}
catch (RuntimeException e) {
    System.out.println("Error running command: " + e.getMessage());
    if (queryRunner.getSession().isDebug()) {
        e.printStackTrace();
    }
}
}

```

从上面的代码可以看出，查询语句的执行分为两部分：初始执行和后续循环发送 RESTful 请求从而分批获取查询结果，这两部分分别是通过方法 `com.facebook.presto.cli.QueryRunner.startQuery(String query)` 和 `com.facebook.presto.cli.Query.renderOutput(PrintStream out, OutputFormat outputFormat, boolean interactive)` 完成的。下面主要针对这两种方法进行源码解析，首先我们对方法 `com.facebook.presto.cli.QueryRunner.startQuery(String query)` 进行解析说明：

```

public Query startQuery(String query) {
    return new Query(startInternalQuery(query));
}

public StatementClient startInternalQuery(String query) {
    return new StatementClient(httpClient, queryResultsCodec, session.get(),
query);
}

public StatementClient(HttpClient httpClient, JsonCodec<QueryResults>
queryResultsCodec, ClientSession session, String query) {
    .....//合法性检查和赋值等操作
    //在 Presto 中是通过向 Coordinator 提供的 RESTful 服务结构发送 HTTP 请求来执行查询的，
下面的方法生成用于发送给 Coordinator 上的 RESTful 服务接口的，发起 SQL 查询的 HTTP 请求
    Request request = buildQueryRequest(session, query);
    JsonResponse<QueryResults> response = httpClient.execute(request,
responseHandler);

    if (response.getStatusCode() != HttpStatus.OK.code())
|| !response.hasValue() {
        throw requestFailedException("starting query", request, response);
    }
    //处理返回的结果
    processResponse(response);
}

private static Request buildQueryRequest(ClientSession session, String query)
{
    // session.getServer() 的内容就是启动 CLI 客户端时通过参数：--server 指定的
Coordinator 的地址，因此其最终生成的 RESTful 的 URI 是一个内容为 CoordinatorIP

```

```

        地址:端口/v1/statement, 类型为 POST 的 RESTful 请求。
        Request.Builder builder = preparePost()
            .setUri(uriBuilderFrom(session.getServer()).replacePath("/v1/sta
            tement").build())
            .setBodyGenerator(createStaticBodyGenerator(query, UTF_8));
        .....//合法性检查和赋值等操作
        return builder.build();
    }

    private void processResponse(JsonResponse<QueryResults> response) {
        .....
        //将返回的结果设置到 Client 端的 currentResults 中, 注意 QueryResults 中含有
        nextResultUri
        currentResults.set(response.getValue());
    }

```

从上面可以看出, 所有的查询请求均由 Coordinator 进行处理, 并将查询结果返回给 Client 端, 那么下面就分析 Coordinator 是如何处理 Client 端发送的 URI 为 CoordinatorIP 地址:端口/v1/statement, 类型为 POST 的 RESTful 请求的。

前面章节已经介绍过了, Statement 的 RESTful 请求均由 StatementResource 类进行处理。对于 URI 为 CoordinatorIP 地址:端口/v1/statement, 类型为 POST 的 RESTful 请求, 由 StatementResource 类中的 createQuery 方法进行处理, 该方法我们已经在前面章节进行了详细介绍。

其次, 我们对方法 com.facebook.presto.cli.Query.renderOutput(PrintStream out, OutputFormat outputFormat, boolean interactive)进行解析说明。该方法在方法 Console.process (QueryRunner queryRunner, String sql, OutputFormat outputFormat, boolean interactive)中, 在方法 Query query = queryRunner.startQuery(sql)之后被调用, 其具体内容如下:

```

public void renderOutput(PrintStream out, OutputFormat outputFormat, boolean
    interactive) {
    Thread clientThread = Thread.currentThread();
    SignalHandler oldHandler = Signal.handle(SIGINT, signal -> {
        if (ignoreUserInterrupt.get() || client.isClosed()) {
            return;
        }
        client.close();
        clientThread.interrupt();
    });
    try {
        //循环获取查询结果, 并动态显示在终端都是在该方法中完成的
        renderQueryOutput(out, outputFormat, interactive);
    }
    finally {
        Signal.handle(SIGINT, oldHandler);
        Thread.interrupted(); // clear interrupt status
    }
}

```

```

private void renderQueryOutput(PrintStream out, OutputFormat outputFormat,

```



```

        boolean interactive) {
    StatusPrinter statusPrinter = null;
    @SuppressWarnings("resource")
    PrintStream errorChannel = interactive ? out : System.err;
    //根据传入的 interactive 标示的内容, 决定是否实时更新结果, 若 interactive 为 true,
    则动态更新结果; 否则不动态更新结果
    if (interactive) {
        statusPrinter = new StatusPrinter(client, out);
        //若需要动态显示查询结果, 则进行间隔打印结果的操作
        statusPrinter.printInitialStatusUpdates();
    }
    else {
        //否则就等待结果
        waitForData();
    }

    //根据 client 的不同状态, 进行不同的后续处理工作。
    if ((!client.isFailed()) && (!client.isGone()) && (!client.isClosed())) {
        .....
    }
}

public void printInitialStatusUpdates()
{
    long lastPrint = System.nanoTime();
    try {
        //当所有数据均已经返回的时候, client.isValid() 的值为 false; 否则其值为 true
        while (client.isValid()) {
            try {
                // 若当前 client 端还有未处理完毕的数据, 就直接退出循环
                if (client.current().getData() != null) {
                    return;
                }

                //若距离上次更新时间超过 0.5 秒, 则打印当前查询的进度信息
                if (Duration.nanosSince(lastPrint).getValue(SECONDS) >= 0.5) {
                    console.repositionCursor();
                    printQueryInfo(client.current());
                    lastPrint = System.nanoTime();
                }

                //获得下一批结果, 注意: 如果 server 端暂时还没有产生数据, 将会等待一段时间
                client.advance();
            }
            catch (RuntimeException e) {
                log.debug(e, "error printing status");
            }
        }
    }
    finally {
        console.resetScreen();
    }
}
}

```



```

private void waitForData() {
    //若 Client 有效且 Client 端的数据被处理完毕了,就循环获得下一批结果,注意:如果 server
    端暂时还没有产生数据,将会等待一段时间
    while (client.isValid() && (client.current().getData() == null)) {
        client.advance();
    }
}

```

从上面的代码可以看到,不管是否动态打印结果,最终都要执行一个方法 `client.advance()`。那么该方法到底做了什么事情呢?该方法的细节如下:

```

//该方法通过使用由 coordinator 返回的 nextResultUri 向 Coordinator 发送请求,来分批获得查
询执行结果
public boolean advance()
{
    //获得请求下一批结果的 URI,该 URI 是由 Coordinator 生成,并返回给 Client 端的,在前
    面提交查询的代码处,对 nextUri 的生成已经进行了讲解,该 URI 的内容为:
    /v1/statement/queryID/token,该 URI 对应的 RestFul 请求也是由 StatementResource 类的
    getQueryResults 方法处理
    URI nextUri = current().getNextUri();
    //请注意:若所有的查询结果均已经返回,则 coordinator 会将 nextUri 设置为 NULL,此时
    会将 valid 设置为 false,而 valid 用于控制是否继续循环调用 advance 方法
    if (isClosed() || (nextUri == null)) {
        valid.set(false);
        return false;
    }
    .....
    //生成查询请求
    Request request = prepareGet()
        .setHeader(USER_AGENT, USER_AGENT_VALUE)
        .setUri(nextUri)
        .build();
    .....
    JsonResponse<QueryResults> response;
    try {
        response = httpClient.execute(request, responseHandler);
    }
    catch (RuntimeException e) {
        cause = e;
        continue;
    }
    if (response.getStatusCode() == HttpStatus.OK.code() &&
    response.hasValue()) {
        processResponse(response);
        return true;
    }
    .....
}

```

我们从上面的代码中可以看到,Client 端对查询结果的分批查询,最终是通过反复调用

StatementClient 的 advance 方法来实现的，每次调用该方法，都会向 Coordinator 发起 nextResultUri 的 RESTful 请求。该方法的调用堆栈如图 4-1 所示。

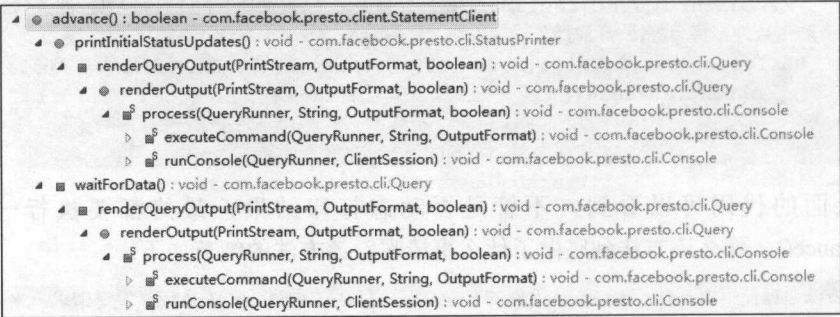


图 4-1

4.3 提交查询的流程

从前面的章节可以看出，提交查询的流程如下所示。

(1) 生成查询的初始 RESTful 请求，并发送给 Coordinator。

(2) Coordinator 处理 RESTful 请求，执行查询，并将部分查询结果和用于 Client 获取下一批查询结果的 RESTful 请求地址 nextResultsURI 一起返回给 Client。

(3) Client 端获得 Coordinator 返回的部分查询结果并进行显示，若所有查询结果均已获取完毕，则直接退出；否则就向 Coordinator 发送 URI 为 nextResultsURI 的 RESTful 请求获取下一批查询执行结果。

(4) 反复执行步骤 (3)，直至所有查询结果均已获取、处理完毕。

查询流程图如图 4-2 所示。

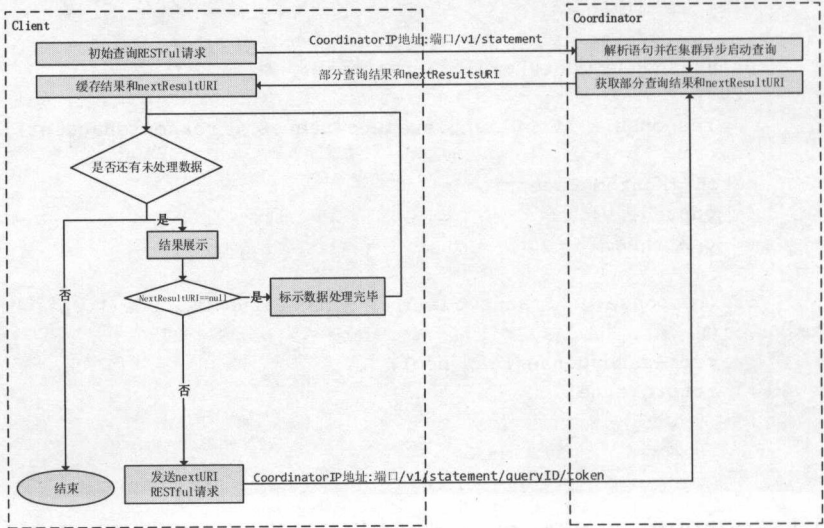


图 4-2

4.4 小结

本章对 Presto 查询提交的步骤和执行流程从源码级别进行了详细的讲解，通过本章的学习，读者可以深入了解客户端如何将查询提交给 Presto 集群，以及如何从 Presto 集群中获得查询执行结果。

第5章

生成查询执行计划

本章主要讲述 Presto 对一个传入的 SQL 语句如何进行解析并生成最终的执行计划。在后续章节中介绍的查询调度与查询执行都是基于生成的执行计划进行的。生成查询执行计划的主要流程如图 5-1 所示。

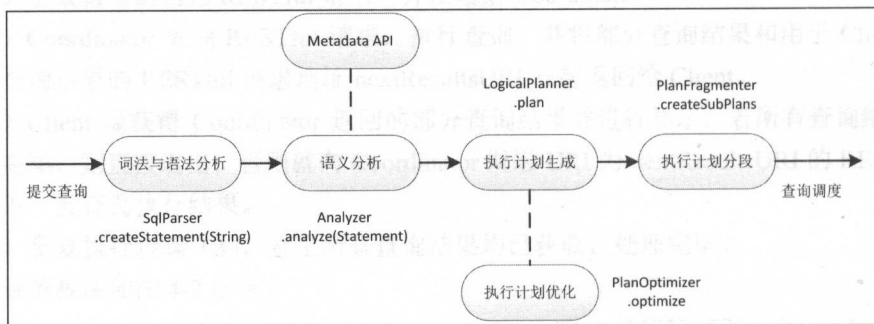


图 5-1

可以看到，从提交查询到执行计划生成包含了词法与语法分析、语义分析、执行计划生成、执行计划优化、执行计划分段等部分。

从 3.1 节中可以看到，Coordinator 处理传入 SQL 查询的入口在类 `StatementResource` 的 `createQuery` 方法中。

```
@POST
@Produces(MediaType.APPLICATION_JSON)
public Response createQuery(
    String statement,
    @Context HttpServletRequest servletRequest,
    @Context UriInfo uriInfo)
    throws InterruptedException
{
    assertRequest(!isEmpty(statement), "SQL statement is empty");
```



```

Session session = createSessionForRequest(servletRequest);

ExchangeClient exchangeClient = exchangeClientSupplier.get();
Query query = new Query(session, statement, queryManager, exchangeClient);
queries.put(query.getQueryId(), query);

return getQueryResults(query, Optional.empty(), uriInfo, new Duration(1,
MILLISECONDS));
}

```

而在 Query 类的构造方法中有如下调用：

```
QueryInfo queryInfo = queryManager.createQuery(session, query);
```

此处的参数 query 即用户传入的 SQL 语句。

QueryManager 是一个接口，而 SQLQueryManager 是唯一实现了该接口的类，因此以上代码实际是调用的 SQLQueryManager 的 createQuery 方法。

5.1 基本概念

首先介绍本章所涉及的一些基本概念。

5.1.1 Node

查询语句经过词法与语法分析之后，会生成抽象语法树（AST），该语法树中的每一个节点都是一个 Node（SQL 语句的一部分，例如 Select 部分、Where 部分、Group by 部分等），Node 是一个抽象类，该类的子类如图 5-2 所示。



图 5-2

下面简单介绍各种类型节点的含义。

1. Approximate

Approximate 用于近似查询。

2. ExplainOption

ExplainOption 表示 Explain 语句中的可选参数, 有 ExplainFormat 和 ExplainType 两个子类。

ExplainFormat 表示 Explain 语句输出结果的格式, 有 TEXT 和 GRAPHVIZ 两种类型。

ExplainType 表示 Explain 语句的类型, 有 LOGICAL 和 DISTRIBUTED 两种类型, 分别表示生成逻辑执行计划与生成分布式执行计划。

3. Expression

Expression 表示 SQL 语句中出现的表达式, 表达式的类型如图 5-3 所示。



图 5-3

4. FrameBound

FrameBound 用于窗口函数中滑动窗口的可选参数。

5. Relation

Relation 是一个抽象类，其子类如图 5-4 所示。



图 5-4

可见这种类型的节点大多都包含了多个其他节点，或者表示多个节点之间的关系，如 Join、Union 等。

6. Select

Select 表示查询语句中的 Select 部分。该类主要元素有：

```
private final boolean distinct; // 是否含有 distinct 关键字
private final List<SelectItem> selectItems; // select 的列
```

7. SelectItem

SelectItem 表示 Select 语句中的列类型，有 AllColumns 和 SingleColumn 两种类型。AllColumns 表示选取所有列，即 Select *。SingleColumn 表示单一列。

8. SortItem

SortItem 表示排序的某一列及其类型，该类的主要元素有：

```
private final Expression sortKey; // 表示排序的列或表达式
private final Ordering ordering; // 表示排序类型，有升序 (ASC) 和降序 (DESC) 两种类型
private final NullOrdering nullOrdering;
```

其中 NullOrdering 包含 First 和 Last 两种类型，分别表示在排序后 NULL 值放在非 NULL 值的前面和后面。

9. Statement

Statement 表示 Presto 中所能使用的所有类型的 SQL 语句，Statement 类的子类如图 5-5 所示。



图 5-5

10. TableElement

TableElement 用于表示建表语句中描述表的每一列，包括列名与类型。

11. Window

Window 表示一个窗口函数，其主要元素有：

```
private final List<Expression> partitionBy; // partition by 的语句或表达式
private final List<SortItem> orderBy; // order by 的语句或表达式
private final Optional<WindowFrame> frame; // 滑动窗口的可选参数
```

12. WindowFrame

WindowFrame 表示窗口函数中滑动窗口的可选参数。

13. With

With 表示一个查询中所有的 With 语句，主要元素有：

```
private final boolean recursive; // 表示 With 语句是否是递归的
private final List<WithQuery> queries; // 表示所有的 With 语句
```

With 语句在一次 SQL 查询中对子查询命名，以使查询语句的结构更清晰，例如以下两个查询是等效的：

```
SELECT a, b
FROM (
  SELECT a, MAX(b) AS b FROM t GROUP BY a
) AS x;

WITH x AS (SELECT a, MAX(b) AS b FROM t GROUP BY a)
```



```
SELECT a, b FROM x;
```

14. WithQuery

WithQuery 表示一个 With 语句，主要元素有：

```
private final String name; // With 语句的别名
private final Query query; // With 所关联的查询语句
private final List<String> columnNames; // With 语句中的列名
```

5.1.2 Metadata API

如图 5-1 所示，在语义分析过程中用到了 Metadata API，以下对其进行详细讲解。

Metadata API 即 Metadata 接口，其提供了对元数据进行各种操作的接口，例如列出所有的库名（listSchemaNames）、表名（listTables）等。这些接口在对 SQL 进行语义分析以及某些 DDL 操作（如 create table）的执行过程中会用到。

Metadata API 将不同 Connector 对其元数据的各种操作抽象成了统一的接口，使得在使用这些接口时无须考虑具体的底层 Connector 实现。

Metadata API 除了提供对元数据操作的接口，还提供了一些通用的与 Connector 无关的方法，例如列出 Presto 支持的自定义函数（listFunctions）等。

Metadata API 如图 5-6 所示。

```
Metadata
  * verifyComparableOrderableContract(): void
  * getType(TypeSignature): Type
  * resolveFunction(QualifiedName, List<TypeSignature>, boolean): FunctionInfo
  * getExactFunction(Signature): FunctionInfo
  * isAggregationFunction(QualifiedName): boolean
  * listFunctions(): List<ParametricFunction>
  * addFunctions(List<? extends ParametricFunction>): void
  * resolveOperator(OperatorType, List<? extends Type>): FunctionInfo
  * listSchemaNames(Session, String): List<String>
  * getTableHandle(Session, QualifiedTableName): Optional<TableHandle>
  * getLayouts(Session, TableHandle, Constraint<ColumnHandle>, Optional<Set<ColumnHandle>>): List<TableLayoutResult>
  * getLayout(Session, TableLayoutHandle): TableLayout
  * getTableMetadata(Session, TableHandle): TableMetadata
  * listTables(Session, QualifiedTablePrefix): List<QualifiedTableName>
  * getSampleWeightColumnHandle(Session, TableHandle): Optional<ColumnHandle>
  * canCreateSampledTables(Session, String): boolean
  * getColumnHandles(Session, TableHandle): Map<String, ColumnHandle>
  * getColumnMetadata(Session, TableHandle, ColumnHandle): ColumnMetadata
  * listTableColumns(Session, QualifiedTablePrefix): Map<QualifiedTableName, List<ColumnMetadata>>
  * createTable(Session, String, TableMetadata): void
  * renameTable(Session, TableHandle, QualifiedTableName): void
  * renameColumn(Session, TableHandle, ColumnHandle, String): void
  * dropTable(Session, TableHandle): void
  * beginCreateTable(Session, String, TableMetadata): OutputTableHandle
  * commitCreateTable(Session, OutputTableHandle, Collection<Slice>): void
  * rollbackCreateTable(Session, OutputTableHandle): void
  * beginInsert(Session, TableHandle, InsertOption): InsertTableHandle
  * commitInsert(Session, InsertTableHandle, Collection<Slice>): void
  * rollbackInsert(Session, InsertTableHandle): void
  * getUpdateRowIdColumnHandle(Session, TableHandle): ColumnHandle
  * beginDelete(Session, TableHandle): TableHandle
  * commitDelete(Session, TableHandle, Collection<Slice>): void
  * rollbackDelete(Session, TableHandle): void
  * getCatalogNames(): Map<String, String>
  * listViews(Session, QualifiedTablePrefix): List<QualifiedTableName>
  * getView(Session, QualifiedTableName): Optional<ViewDefinition>
  * createView(Session, QualifiedTableName, String, boolean): void
  * dropView(Session, QualifiedTableName): void
  * getFunctionRegistry(): FunctionRegistry
  * getTableManager(): TableManager
  * getBlockEncodingSerde(): BlockEncodingSerde
  * getSessionPropertyManager(): SessionPropertyManager
  * getTablePropertyManager(): TablePropertyManager
```

图 5-6

Metadata 接口唯一的实现类是 MetadataManager，该类中对元数据操作的接口的实现使用了 ConnectorMetadata 接口。ConnectorMetadata 接口只包含 Metadata API 中元数据操作的接口。该接口中的方法描述可参考 12.4 节。

ConnectorMetadata 接口如图 5-7 所示。



图 5-7

不同的 Connector 都实现了 ConnectorMetadata 接口，如图 5-8 所示。

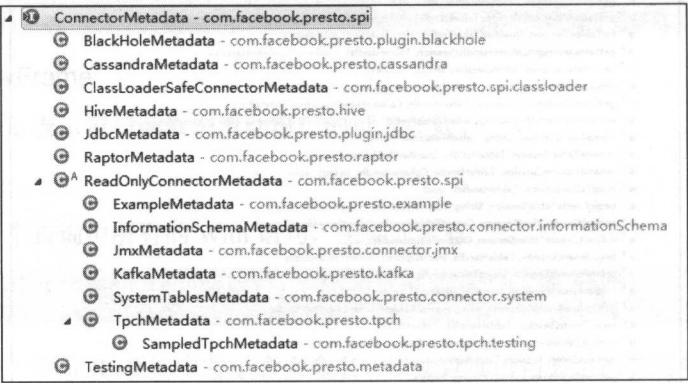


图 5-8

5.2 词法与语法分析

Presto 的词法与语法分析的入口在 SQLQueryManager 的 createQuery 方法中：

```
QueryId queryId = queryIdGenerator.createNextQueryId();

Statement statement;
QueryExecutionFactory<?> queryExecutionFactory;
try {
    statement = sqlParser.createStatement(query); // 词法与语法分析获取
Statement
    queryExecutionFactory = executionFactories.get(statement.getClass());
    ... ..
}
```

5.2.1 语法规则

Presto 使用 ANTLR4 编写 SQL 语法,语法规则的定义在 presto-parser 项目的 SqlBase.g4 文件中,通过 Eclipse 的 ANTLR4 插件可以查看该文件的语法图,如图 5-9 所示。

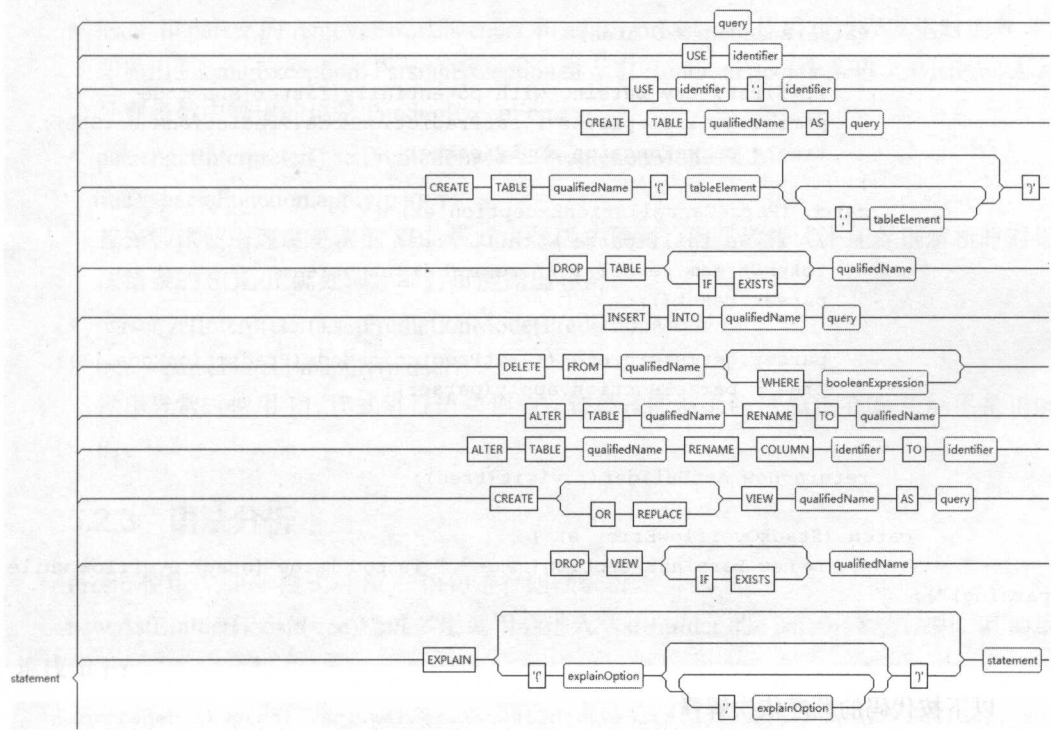


图 5-9

其中每一条从左到右的支线都表示一种类型的 SQL 语句,即 5.1.1 节中的 Statement。而如 DROP TABLE 语句中的分支表示,在 DROP TABLE 和表名之间可以有 IF EXISTS 关键字,也可以什么关键字都不加。

5.2.2 词法分析

SQLParser 的 createStatement 方法调用其内部方法 invokeParser:

```
private Node invokeParser(String name, String sql, Function<SqlBaseParser,
```

```

ParserRuleContext> parseFunction()
{
    try {
        SqlBaseLexer lexer = new SqlBaseLexer(new CaseInsensitiveStream(new
ANTLRInputStream(sql)));
        CommonTokenStream tokenStream = new CommonTokenStream(lexer);
        SqlBaseParser parser = new SqlBaseParser(tokenStream);

        parser.addParseListener(new PostProcessor());

        lexer.removeErrorListeners();
        lexer.addErrorListener(ERROR_LISTENER);

        parser.removeErrorListeners();
        parser.addErrorListener(ERROR_LISTENER);

        ParserRuleContext tree;
        try {
            // first, try parsing with potentially faster SLL mode
            parser.getInterpreter().setPredictionMode(PredictionMode.SLL);
            tree = parseFunction.apply(parser);
        }
        catch (ParseCancellationException ex) {
            // if we fail, parse with LL mode
            tokenStream.reset(); // rewind input stream
            parser.reset();

            parser.getInterpreter().setPredictionMode(PredictionMode.LL);
            tree = parseFunction.apply(parser);
        }

        return new AstBuilder().visit(tree);
    }
    catch (StackOverflowError e) {
        throw new ParsingException(name + " is too large (stack overflow while
parsing)");
    }
}

```

以下按代码的顺序依次解释。

- **SqlBaseLexer 类和 SqlBaseParser 类**

它们是通过 ANTLR4 编译 `SqlBase.g4` 生成的类。

- **CaseInsensitiveStream 类**

其是为了使输入的 SQL 语句大小写不敏感,即忽略 SQL 语句关键字的大小写并视为等效的关键字,例如将 `Select`、`select` 与 `SELECT` 视为等效关键字。

- **parser.addParseListener(new PostProcessor())**

这是为了做一些解析时的处理,包括一些异常处理。`PostProcessor` 的处理函数有:

- **exitUnquotedIdentifier**

如果未用引号括起来的标识符中有 “@” 或 “.” 等符号则抛出异常。

- `exitBackQuotedIdentifier`

如果标识符是用反引号“`”括起来的则抛出异常。

- `exitDigitIdentifier`

如果标识符是以数字开头的则抛出异常。

- `exitQuotedIdentifier`

对于双引号引起来的标识符，去除双引号。例如 `end` 为关键字，在查询时如果其为标识符则需要使用双引号将其括起来：`SELECT "end"...`，而在解析后将双引号去除。

- `exitNonReserved`

将非保留关键字替换成标识符，以使处理时将其当作标识符，例如 `SELECT show ...`，这里将 `show` 当作一个标识符，而不是当作关键字进行处理。

- `lexer` 和 `parser` 的 `removeErrorListeners` 和 `addErrorListener` 是重写错误发生时的处理，即抛出 `ParsingException`，`ParsingException` 继承自 `RuntimeException`，以上 `PostProcessor` 处理函数中抛出的也是 `ParsingException`。

- `parser.getInterpreter().setPredictionMode(PredictionMode.SLL)`

`tree = parseFunction.apply(parser)`

首先尝试使用速度更快的 SLL 模式进行语法预测，但是该模式不能保证解析时对语法错误的 SQL 正确处理，即有可能抛出异常。

- `parser.getInterpreter().setPredictionMode(PredictionMode.LL)`

`tree = parseFunction.apply(parser)`

抛出异常时使用 LL 模式进行语法预测，该模式能够确保对 SQL 的解析结果是正确的。

5.2.3 语法分析

Presto 使用 Visitor 模式对 SQL 语句进行语法分析。

`new AstBuilder().visit(tree)` 经过多层调用会进入 `AstBuilder` 的 `visitXXX` 方法中，其调用堆栈如下：

```
com.facebook.presto.sql.parser.AstBuilder.visitSingleStatement(AstBuilder.java:124)
com.facebook.presto.sql.parser.SqlBaseParser$SingleStatementContext.accept(SqlBaseParser.java:139)
org.antlr.v4.runtime.tree.AbstractParseTreeVisitor.visit(AbstractParseTreeVisitor.java:44)
```

`AstBuilder.visitSingleStatement` 方法如下：

```
public Node visitSingleStatement(@NotNull SqlBaseParser.SingleStatementContext context)
{
    return visit(context.statement());
}
```

visit(context.statement())方法会根据 SQL 类型调用对应的 visit×××方法，以 INSERT 语法为例，该方法会调用 visitInsertInto 方法：

```
public Node visitInsertInto(@NotNull SqlBaseParser.InsertIntoContext context)
{
    return new Insert(getQualified_name(context.qualified_name()), (Query)
visit(context.query()));
}
```

可见，该函数会构造一个 Insert 对象，该对象包含 INSERT 的目的表名，以及一个 Query 对象，该 Query 对象也是通过 visit(context.query())调用 AstBuilder.visitQuery 方法构造的。因此对一个 Statement 的语法分析实际上是一个递归调用的过程，构造一个个的节点，最终形成一个抽象语法树，该语法树实际上就是一个 Statement。

至此，经过对 SQL 语句的词法和语法分析，根据 SQL 语句的类型返回了一个 Statement 的子类对象。

5.3 获取 QueryExecution

QueryExecution 表示一次查询执行，用于启动、停止与管理一个查询，以及统计这个查询的相关信息。QueryExecution 是一个接口，其实现类有 DataDefinitionExecution、SqlQueryExecution 和 FailedQueryExecution。

5.3.1 获取 QueryExecutionFactory

回到 SqlQueryManager 的 createQuery 方法：

```
statement = sqlParser.createStatement(query);
```

根据以上获取的 statement 类型，获取相对应的 QueryExecutionFactory：

```
queryExecutionFactory = executionFactories.get(statement.getClass());
```

QueryExecutionFactory 是一个接口，其实现类有 DataDefinitionExecutionFactory 以及 SqlQueryExecutionFactory。

executionFactories 则是一个 Map，存储了不同的 Statement 类型与 QueryExecutionFactory 实现类的对应关系，该 Map 的初始化是在 CoordinatorModule 中进行的，对应关系如表 5-1 所示。

表 5-1

Statement 实现类	QueryExecutionFactory 实现类
CreateTable	DataDefinitionExecutionFactory
RenameTable	
RenameColumn	
DropTable	
CreateView	

续表

Statement 实现类	QueryExecutionFactory 实现类
DropView	
SetSession	
ResetSession	
Query	SqlQueryExecutionFactory
Explain	
ShowColumns	
ShowPartitions	
ShowFunctions	
ShowTables	
ShowSchemas	
ShowCatalogs	
Use	
ShowSession	
CreateTableAsSelect	
Insert	
Delete	

Create Table 、 Rename Table 等 DDL 操作的 SQL 语句对应了 DataDefinitionExecutionFactory；而非 DDL 操作的 SQL 语句，例如 Select、Insert 等对应了 SqlQueryExecutionFactory。

该 Map 是通过 Guice 注入传递到 SqlQueryManager 中的。

5.3.2 创建 QueryExecution

1. 错误处理

当以上的词法与语法分析出错，或者找不到 Statement 实现类与 QueryExecutionFactory 实现类的对应关系时，将创建一个 FailedQueryExecution，并封装错误信息，最后返回给用户。

2. 构造 QueryExecution

调用之前获取的 QueryExecutionFactory 的 createQueryExecution 方法，获取对应的 QueryExecution。DataDefinitionExecutionFactory 创建的是 DataDefinitionExecution，而 SqlQueryExecutionFactory 创建的是 SqlQueryExecution。

在 DataDefinitionExecutionFactory 创建 DataDefinitionExecution 时，根据 Statement 类型将对应的 DataDefinitionTask 实现类与 DataDefinitionExecution 绑定：

```
public DataDefinitionExecution<?> createQueryExecution(  
    QueryId queryId,  
    String query,
```

```
        Session session,
        Statement statement)
    {
        URI self = locationFactory.createQueryLocation(queryId);
        QueryStateMachine stateMachine = new QueryStateMachine(queryId, query, session,
self, executor);
        return createExecution(statement, session, stateMachine);
    }

    private <T extends Statement> DataDefinitionExecution<?> createExecution(
        T statement,
        Session session,
        QueryStateMachine stateMachine)
    {
        DataDefinitionTask<T> task = getTask(statement);
        checkArgument(task != null, "no task for statement: %s",
statement.getClass().getSimpleName());
        stateMachine.setUpdateType(task.getName());
        return new DataDefinitionExecution<>(task, statement, session, metadata,
stateMachine);
    }

    private <T extends Statement> DataDefinitionTask<T> getTask(T statement)
    {
        return (DataDefinitionTask<T>) tasks.get(statement.getClass());
    }
}
```

其中 tasks 定义在 DataDefinitionExecutionFactory 类中：

```
private final Map<Class<? extends Statement>, DataDefinitionTask<?>> tasks;
```

该 Map 的初始化同样在 CoordinatorModule 中进行，对应关系如表 5-2 所示。

表 5-2

Statement 实现类	DataDefinitionTask 实现类
CreateTable	CreateTableTask
RenameTable	RenameTableTask
RenameColumn	RenameColumnTask
DropTable	DropTableTask
CreateView	CreateViewTask
DropView	DropViewTask
SetSession	SetSessionTask
ResetSession	ResetSessionTask

5.3.3 启动 QueryExecution

获取 QueryExecution 之后，SqlQueryQueueManager.submit 方法将 QueryExecution 与配置的查询队列规则进行匹配，若匹配成功且队列未满，则将 QueryExecution 加入匹配队列。查询队列按照 FIFO 规则调度查询，关于队列的配置可参考第 8 章。

最后启动 QueryExecution, 下面对 QueryExecution 的两个实现类进行讲解。

1. 启动 DataDefinitionExecution

DataDefinitionExecution 的启动方法如下:

```
public void start()
{
    try {
        // transition to running
        if (!stateMachine.transitionToRunning()) {
            // query already running or finished
            return;
        }
        task.execute(statement, session, metadata, stateMachine);
        stateMachine.transitionToFinished();
    }
    catch (Throwable e) {
        fail(e);
        if (!(e instanceof RuntimeException)) {
            throw Throwables.propagate(e);
        }
    }
}
```

可见 DataDefinitionExecution 的启动比较简单, 直接调用与其绑定的 DataDefinitionTask 实现类的 execute 方法即可。

以 DropTable 为例, 由于 DropTable 与 DropTableTask 绑定, 会直接执行 DropTableTask 的 execute 方法:

```
public void execute(DropTable statement, Session session, Metadata metadata,
QueryStateMachine stateMachine)
{
    QualifiedTableName tableName = createQualifiedTableName(session,
statement.getTableName());
    Optional<TableHandle> tableHandle = metadata.getTableHandle(session,
tableName);
    if (!tableHandle.isPresent()) {
        if (!statement.isExists()) {
            throw new SemanticException(MISSING_TABLE, statement, "Table '%s' does
not exist", tableName);
        }
        return;
    }
    metadata.dropTable(tableHandle.get());
}
```

从上面的代码可以看出, 首先调用 Metadata 的 getTableHandle 方法, 根据表名获取 TableHandle, 如果 DROP 的表不存在且 SQL 语句中没有 IF NOT EXISTS 判断, 则抛出异常; 如果表存在, 则调用 Metadata API 的 dropTable 方法删除表。

2. 启动 SqlQueryExecution

SqlQueryExecution 的启动方法如下：

```
public void start()
{
    try (SetThreadName ignored = new SetThreadName("Query-%s", stateMachine.
getQueryId())) {
        try {
            // transition to planning
            if (!stateMachine.transitionToPlanning()) {
                // query already started or finished
                return;
            }
            // analyze query
            SubPlan subplan = analyzeQuery();
            // plan distribution of query
            planDistribution(subplan);
            // transition to starting
            if (!stateMachine.transitionToStarting()) {
                // query already started or finished
                return;
            }
            // if query is not finished, start the stage, otherwise cancel it
            SqlStageExecution stage = outputStage.get();
            if (!stateMachine.isDone()) {
                stage.start();
            }
        }
        catch (Throwable e) {
            fail(e);
            Throwables.propagateIfInstanceOf(e, Error.class);
        }
    }
}
```

SubPlan subplan = analyzeQuery()即生成查询执行计划，函数调用层次为 analyzeQuery → doAnalyzeQuery。doAnalyzeQuery 实现如下：

```
private SubPlan doAnalyzeQuery()
{
    // time analysis phase
    long analysisStart = System.nanoTime();

    // analyze query
    Analyzer analyzer = new Analyzer(stateMachine.getSession(), metadata,
sqlParser, Optional.of(queryExplainer), experimentalSyntaxEnabled);
    Analysis analysis = analyzer.analyze(statement);

    stateMachine.setUpdateType(analysis.getUpdateType());

    // plan query
    PlanNodeIdAllocator idAllocator = new PlanNodeIdAllocator();
    LogicalPlanner logicalPlanner = new LogicalPlanner(stateMachine.getSession(),
```

```

planOptimizers, idAllocator, metadata);
    Plan plan = logicalPlanner.plan(analysis);

    // extract inputs
    List<Input> inputs = new InputExtractor(metadata).extract(plan.getRoot());
    stateMachine.setInputs(inputs);

    // fragment the plan
    SubPlan subplan = new PlanFragmenter().createSubPlans(plan);

    // record analysis time
    stateMachine.recordAnalysisTime(analysisStart);
    return subplan;
}

```

其中 `analyzer.analyze(statement)` 对传入的 SQL 进行语义分析, `logicalPlanner.plan(analysis)` 针对语义分析的结果生成查询执行计划。

5.4 语义分析

由于 `DataDefinitionExecution` 的执行直接调用 `DataDefinitionTask` 实现类的 `execute` 方法, 并未经过执行计划生成的步骤, 故以下内容只针对 `SqlQueryExecutionFactory` 所对应的 `Statement` 实现类, 见表 5-1。

`Analyzer` 的 `analyze` 方法如下:

```

public Analysis analyze(Statement statement)
{
    Analysis analysis = new Analysis();
    StatementAnalyzer analyzer = new StatementAnalyzer(analysis, metadata,
sqlParser, session, experimentalSyntaxEnabled, queryExplainer);
    TupleDescriptor outputDescriptor = analyzer.process(statement, new
AnalysisContext());
    analysis.setOutputDescriptor(outputDescriptor);
    return analysis;
}

```

该方法主要构造了一个 `StatementAnalyzer` 对传入的 `Statement` 进行分析, 将分析结果存储在 `Analysis` 中并返回。

5.4.1 Statement 分析

`StatementAnalyzer` 是对 `Statement` 进行语义分析的类, 针对不同的 `Statement` 实现类进行了语义分析, 该类的类视图如图 5-10 所示。

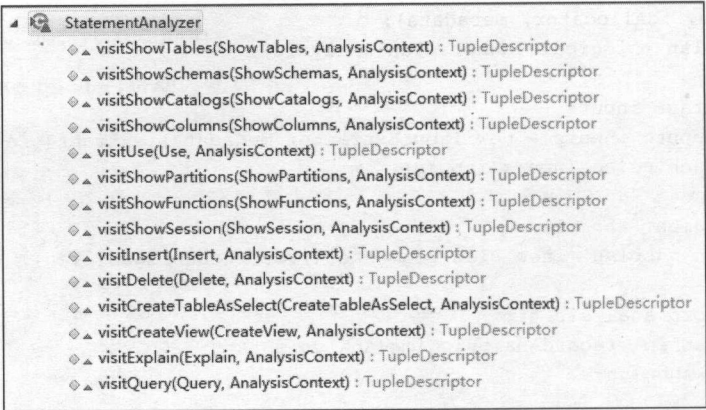


图 5-10

StatementAnalyzer 对 Statement 进行语义分析后返回的结果为 SQL 语句所输出的列描述符，是 TupleDescriptor 类型的，一个 TupleDescriptor 实例的结构如图 5-11 所示。

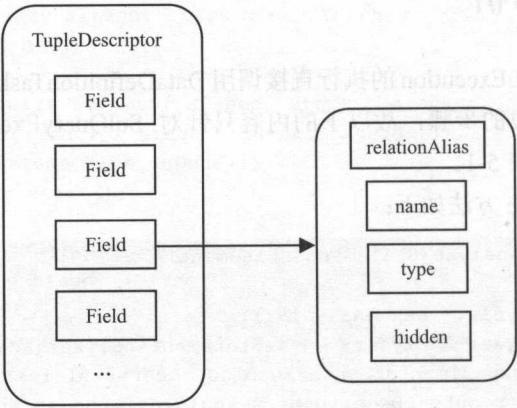


图 5-11

可见 TupleDescriptor 包含了一系列的 Field，一个 Field 表示对一个字段的描述，每个 Field 包含字段名称 name、字段别名 relationAlias、字段类型 type、字段是否隐藏 hidden。

对于 Select 语句和 Show ××× 语句，返回的 TupleDescriptor 包含取到的每一列，每一个 Field 表示一列；对于 Insert 语句、Delete 语句和 Create Table As Select 语句，返回的 TupleDescriptor 只有一列，表示语句执行所操作的行数。

StatementAnalyzer 同样使用 Visitor 模式对 Statement 进行语义分析，该类中的各个 Visit ××× 方法针对不同的 Statement 实现类做了处理，以下针对不同的 Statement 分类分别进行讲解。

1. 元信息相关命令分析

该类命令主要用于查看集群中的各种元信息，包括 Show Tables、Show Schemas、Show Catalogs、Show Columns、Show Partitions、Show Functions、Show Session，以及用于切换

库的 USE 命令。

(1) information_schema 查询

Presto 在启动时为每个 Catalog 都建立了一个 information_schema 库，其中有 6 张表：

- __internal_functions__
该表存储了 Presto 内部注册的所有函数。
- __internal_partitions__
该表存储了当前 Catalog 下所有分区表的分区信息。
- columns
该表存储了当前 Catalog 下所有表的列信息。
- schemata
该表存储了当前 Catalog 下所有的库信息。
- tables
该表存储了当前 Catalog 下所有的表信息。
- views
该表存储了当前 Catalog 下所有的视图信息。

StatementAnalyzer 中 visitShowTables 的实际操作是组装了一个 Select 查询，查询 information_schema 的 tables 表，然后通过 StatementAnalyzer 的 visitQuery 进行分析。以下给出查询命令以及与其对应的 information_schema 中的表的对应关系，表 5-3 中的查询命令都会转换成对应的 Select 查询语句去查询对应的表。

表 5-3	
查询命令	对应表
Show Tables	tables
Show Schemas	schemata
Show Columns	columns
Show Partitions	<u>__internal_partitions__</u>
Show Functions	<u>__internal_functions__</u>

(2) Show Catalogs

Show Catalogs 显示的是当前集群所注册的 Catalog，对应的处理方法为 visitShowCatalogs，该方法先获取集群中的 Catalog 列表，然后组装成如下的一个 SELECT 查询：

```
SELECT * FROM (VALUES ('hive'), ('mysql'), ('kafka')) catalogs(Catalog);
```

该查询同样会通过 StatementAnalyzer 的 visitQuery 进行分析处理。

(3) Show Sessions

Show Sessions 显示的是当前 session 中所有的 session 参数，对应的处理方法为 visitShowSession，与 Show Catalogs 类似，该方法也是先获取当前所有的 session 参数，然后组装成如下的一个 Select 查询：

```
SELECT name Name, value Value
FROM
(VALUES ('distributed_join', 'true'), ('task_writer_count', '4'))
sessions(name, value);
```

该查询同样会通过 StatementAnalyzer 的 visitQuery 进行分析处理。

(4) Use 命令

Presto 的任何查询都属于一个 session，而每个 session 都是有自己的 Catalog 和 Schema 的。其中 Catalog 需要在连接 Presto 的时候指出，Schema 也可以指出，若不指出则为 default。

USE 命令就是用来切换 Catalog 和 Schema 的。

切换同一个 Catalog 下的 Schema 的命令为：Use schema1。

切换至不同 Catalog 下的 Schema 的命令为：Use otherCatalog.schema2。

目前 Use 命令只在 Cli 连接 Presto 时有用，做法是更新客户端 Session 的 Catalog 和 Schema，如果使用 JDBC 等其他方式连接 Presto，则不支持该语法，StatementAnalyzer 的 visitUse 方法会报语义错误 “USE statement is not supported”。

2. Create View

目前只有 Hive 和 Raptor 两个 Connector 支持创建视图操作，其余的 Connector 中如果执行 Create View 命令，则会抛出异常。

visitCreateView 方法如下：

```
protected TupleDescriptor visitCreateView(CreateView node, AnalysisContext
context)
{
    analysis.setUpdateType("CREATE VIEW");
    TupleDescriptor descriptor = process(node.getQuery(), context);
    validateColumnNames(node, descriptor);
    return descriptor;
}
```

可见对 Create View 命令处理时，先对 Create View 所附带的查询语句进行分析，对返回的 TupleDescriptor 进行校验，validateColumnNames 方法确保查询语句的每一列都有列名且列名唯一，该方法如下：

```
private static void validateColumnNames(Statement node, TupleDescriptor
descriptor)
{
    // verify that all column names are specified and unique
    Set<String> names = new HashSet<>();
    for (Field field : descriptor.getVisibleFields()) {
        Optional<String> fieldName = field.getName();
        if (!fieldName.isPresent()) {
            throw new SemanticException(COLUMN_NAME_NOT_SPECIFIED, node, "Column
name not specified at position %s", descriptor.indexOf(field) + 1);
        }
        if (!names.add(fieldName.get())) {
            throw new SemanticException(DUPLICATE_COLUMN_NAME, node, "Column name
```

```
'%s' specified more than once", fieldName.get());
    }
}
}
```

从 5.3.2 节可以看到 Create View 操作的处理在 CreateViewTask 中，处理时调用了 Analyzer 的 analyze 方法，因此会在 StatementAnalyzer 中进行部分处理，后续会调用对应 Connector 的 createView 方法进行处理，当然以上也提到了，只有 Hive Connector 和 Raptor Connector 具体实现了 Create View 操作，其他的 Connector 则直接抛出异常。

3. Create Table As Select

visitCreateTableAsSelect 方法如下：

```
protected TupleDescriptor visitCreateTableAsSelect(CreateTableAsSelect node,
AnalysisContext context)
{
    analysis.setUpdateType("CREATE TABLE");
    // turn this into a query that has a new table writer node on top.
    QualifiedTableName targetTable = MetadataUtil.createQualifiedTableName
(session, node.getName());
    analysis.setCreateTableDestination(targetTable);
    Optional<TableHandle> targetTableHandle = metadata.getTableHandle(session,
targetTable);
    if (targetTableHandle.isPresent()) {
        throw new SemanticException(TABLE_ALREADY_EXISTS, node, "Destination table
's' already exists", targetTable);
    }
    // analyze the query that creates the table
    TupleDescriptor descriptor = process(node.getQuery(), context);
    validateColumnNames(node, descriptor);
    return new TupleDescriptor(Field.newUnqualified("rows", BIGINT));
}
```

可见处理 Create Table As Select 语句时首先是获取目的表的 TableHandle，如果表已存在，则抛出异常，否则分析查询语句，获取 TupleDescriptor，同样用 validateColumnNames 方法校验查询语句的列，确保查询语句的每一列都有列名且列名唯一。

4. Insert

visitInsert 方法如下：

```
protected TupleDescriptor visitInsert(Insert insert, AnalysisContext context)
{
    analysis.setUpdateType("INSERT");

    // analyze the query that creates the data
    TupleDescriptor descriptor = process(insert.getQuery(), context);

    // verify the insert destination columns match the query
    QualifiedTableName targetTable = MetadataUtil.
createQualifiedTableName(session, insert.getTarget());
```



```

Optional<TableHandle> targetTableHandle = metadata.getTableHandle(session,
targetTable);
if (!targetTableHandle.isPresent()) {
    throw new SemanticException(MISSING_TABLE, insert, "Table '%s' does not
exist", targetTable);
}
analysis.setInsertTarget(targetTableHandle.get());

List<ColumnMetadata> columns = metadata.getTableMetadata(targetTableHandle.
get()).getColumns();
Iterable<Type> tableTypes = FluentIterable.from(columns)
    .filter(column -> !column.isHidden())
    .transform(ColumnMetadata::getType);
Iterable<Type> queryTypes = transform(descriptor.getVisibleFields(),
Field::getType);
if (!elementsEqual(tableTypes, queryTypes)) {
    throw new SemanticException(MISMATCHED_SET_COLUMN_TYPES, insert, "Insert
query has mismatched column types: " +
        "Table: (" + Joiner.on(", ").join(tableTypes) + "), " +
        "Query: (" + Joiner.on(", ").join(queryTypes) + ")");
}
return new TupleDescriptor(Field.newUnqualified("rows", BIGINT));
}

```

可见处理 Insert 语句时，首先分析 Insert 所关联的查询语句，然后获取目的表的 TableHandle。如果表不存在，则报错；如果表存在，则校验查询语句和目的表的每一列对应的类型是否相同，如果存在列的类型不对应，则抛出异常。

5. Delete

visitDelete 方法如下：

```

protected TupleDescriptor visitDelete(Delete node, AnalysisContext context)
{
    analysis.setUpdateType("DELETE");
    analysis.setDelete(node);

    TupleAnalyzer analyzer = new TupleAnalyzer(analysis, session, metadata,
sqlParser, experimentalSyntaxEnabled);
    TupleDescriptor descriptor = analyzer.process(node.getTable(), context);
    node.getWhere().ifPresent(where -> analyzer.analyzeWhere(node, descriptor,
context, where));
    return new TupleDescriptor(Field.newUnqualified("rows", BIGINT));
}

```

可见 visitDelete 使用 TupleAnalyzer 对 Delete 语句的目的表进行分析，如果 Delete 指定了 Where 条件，则同样使用 TupleAnalyzer 的 analyzeWhere 方法对其进行分析。

TupleAnalyzer 类会在 5.4.2 节进行分析。

6. Query

visitQuery 方法如下：


```

protected TupleDescriptor visitQuery(Query node, AnalysisContext parentContext)
{
    AnalysisContext context = new AnalysisContext(parentContext);

    if (node.getApproximate().isPresent()) {
        if (!experimentalSyntaxEnabled) {
            throw new SemanticException(NOT_SUPPORTED, node, "approximate queries
are not enabled");
        }
        context.setApproximate(true);
    }
    analyzeWith(node, context);

    TupleAnalyzer analyzer = new TupleAnalyzer(analysis, session, metadata,
sqlParser, experimentalSyntaxEnabled);
    TupleDescriptor descriptor = analyzer.process(node.getQueryBody(), context);
    analyzeOrderBy(node, descriptor, context);

    // Input fields == Output fields
    analysis.setOutputDescriptor(node, descriptor);
    analysis.setOutputExpressions(node, descriptorToFields(descriptor));
    analysis.setQuery(node);
    return descriptor;
}

```

可见该方法分为以下几步。

(1) 分析近似查询

Presto 支持近似查询，但是需要开启配置参数 `experimental-syntax-enabled`，该参数默认是 `false`，设置成 `true` 之后才可以执行近似查询，近似查询的语法示例如下：

```
select count(l_orderkey) from lineitem APPROXIMATE AT 80 CONFIDENCE;
```

若未设定 `experimental-syntax-enabled` 为 `true` 并且执行了以上查询，则直接抛出异常。

(2) 分析 With 语句

Presto 支持 With 语句，会先对 With 语句中的查询语句进行分析，分析同样是调用 `visitQuery` 方法。

(3) 分析查询语句

对查询语句的分析在 `TupleAnalyzer` 类中进行，详见 5.4.2 节。

(4) 分析 Order By 语句

针对 `TupleAnalyzer` 未分析的 Order By 语句进行分析主要包括两部分：一是检验 Order By 的列是否在 Select 语句中，二是当 Order By 不是简单的列而是复杂语句时，对语句进行分析。

7. Explain

`visitExplain` 方法如下：

```

protected TupleDescriptor visitExplain(Explain node, AnalysisContext context)
    throws SemanticException

```

```

{
    checkState(queryExplainer.isPresent(), "query explainer not available");
    ExplainType.Type planType = LOGICAL;
    ExplainFormat.Type planFormat = TEXT;
    List<ExplainOption> options = node.getOptions();

    for (ExplainOption option : options) {
        if (option instanceof ExplainType) {
            planType = ((ExplainType) option).getType();
            break;
        }
    }
    for (ExplainOption option : options) {
        if (option instanceof ExplainFormat) {
            planFormat = ((ExplainFormat) option).getType();
            break;
        }
    }

    String queryPlan = getQueryPlan(node, planType, planFormat);

    Query query = simpleQuery(
        selectList(new AllColumns()),
        aliased(
            values(row(new StringLiteral((queryPlan))),
                "plan",
                ImmutableList.of("Query Plan"))));

    return process(query, context);
}

```

Explain 命令用于查看 SQL 的执行计划，该命令有以下两项可选参数。

(1) Type

执行计划类型，包括 Logical 和 Distributed 两类，默认为 Logical。

(2) Format

执行计划显示格式目前支持 Text 和 Graphviz 两类，默认为 Text。

getQueryPlan 方法根据不同的 Format 调用 QueryExplainer 类的 getPlan 或 getGraphvizPlan 方法，获取查询执行计划，最后对执行计划字符串封装一个 Select 查询，该查询将通过 visitQuery 进行分析处理。

QueryExplainer 类获取查询执行计划的过程与本章生成查询执行计划的步骤相同，这里不再赘述。获取查询执行计划之后，通过 PlanPrinter 对查询执行计划进行格式化输出。

5.4.2 Relation 分析

TupleAnalyzer 类是对 Query 中的 Relation 进行分析的类，该类的类视图如图 5-12 所示。

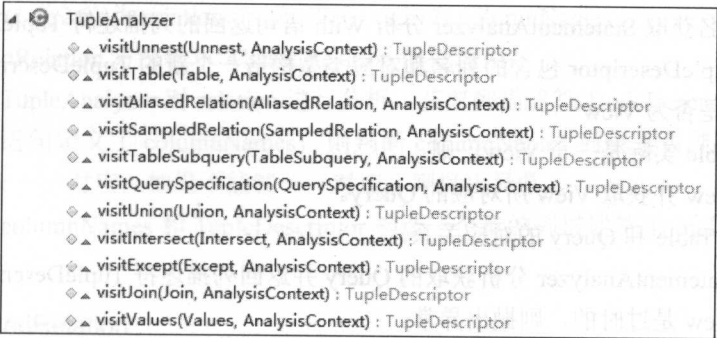


图 5-12

以下针对该类中的方法分别进行讲解。

1. Unnest

Unnest 语句用于将 Array 和 Map 进行展开，示例如下：

```
SELECT numbers, n, a
FROM (
  VALUES
    (ARRAY[2, 5]),
    (ARRAY[7, 8, 9])
) AS x (numbers)
CROSS JOIN UNNEST(numbers) WITH ORDINALITY AS t (n, a);
```

numbers	n	a
[2, 5]	2	1
[2, 5]	5	2
[7, 8, 9]	7	1
[7, 8, 9]	8	2
[7, 8, 9]	9	3

(5 rows)

visitUnnest 方法对 Unnest 语句的处理为：

(1) 分析 Unnest 所使用的列

Unnest 可以使用多列，循环其所使用的列，使用 ExpressionAnalyzer 对其进行分析，获取每一列的类型，如果为 Array 则展开为单列，如果为 Map 则展开为双列。

(2) 分析 With Ordinality

如果 Unnest 之后有 With Ordinality 关键字，则额外增加一列，该列为 Bigint 类型。

2. Table

visitTable 方法对 Table 进行分析包括以下几步。

(1) 分析是否为 With

如果该 Table 是 With 所定义的表名，则：

- 记录下 Table 和 Query 的对应关系。

- 根据表名获取 StatementAnalyzer 分析 With 语句返回的列描述符 TupleDescriptor。
- 依据 TupleDescriptor 包含的列名和对应类型构造一个新的 TupleDescriptor 并返回。

(2) 分析是否为 View

如果该 Table 实际是一个 View，则：

- 解析 View 并获取 View 所对应的 Query。
- 记录下 Table 和 Query 的对应关系。
- 使用 StatementAnalyzer 分析获取的 Query 并返回列描述符 TupleDescriptor。
- 如果 View 是过时的，则抛出异常。
- 返回以上的列描述符 TupleDescriptor。

(3) 分析 Table 是否存在

使用 Metadata 获取 TableHandle，如果 TableHandle 不存在，则：

- 如果 catalog 不存在，则抛出 catalog 不存在的错误消息。
- 如果 schema 不存在，则抛出 schema 不存在的错误消息。
- 如果表名有 DUAL 前缀，则抛出 DUAL 不再支持的错误消息。
- 如果 catalog 和 schema 都存在且表名没有 DUAL 前缀，则抛出表不存在的错误消息。

(4) 构造列描述符 TupleDescriptor

- 根据 TableHandle 获取 TableMetadata 和 ColumnHandle。
- 循环 TableMetadata 的每一个 ColumnMetadata，构造 Field，记录 Field 和 ColumnHandle 的对应关系。
- 记录 TABLE 和 TableHandle 的对应关系。
- 根据 Field 列表构造列描述符 TupleDescriptor 并返回。

3. AliasedRelation

顾名思义，AliasedRelation 就是带别名的 Relation，其结构如图 5-13 所示。

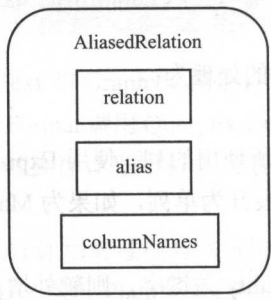


图 5-13

其中 relation 是接口 Relation 的实现，alias 即 relation 的别名，columnNames 是语句中定义的 relation 中列名的列表，该列名列表应该与 relation 返回的列一一对应。

例如在以下 SQL 语句中

```
SELECT * FROM (VALUES (1.0, 2.0)) x (a, b)
```


其中(VALUES (1.0, 2.0))为 relation, x 为别名, (a,b)为 columnNames。

visitAliasedRelation 方法针对 AliasedRelation 的处理为:

- (1) 使用 TupleAnalyzer 对 relation 进行分析, 获取列描述符 TupleDescriptor。
- (2) 如果语句定义了 columnNames, 则判断 columnNames 列表是否能与以上返回的 TupleDescriptor 一一对应, 如果不能一一一对应, 则抛出异常。
- (3) 使用 columnNames 和 TupleDescriptor 构造一个新的列描述符 TupleDescriptor 并返回。

4. SampledRelation

Presto 支持对表进行抽样, 例如:

```
SELECT * FROM users TABLESAMPLE BERNOULLI (50);
```

抽样类型包括 BERNOULLI、POISSONIZED 和 SYSTEM 三种。

visitSampledRelation 方法对抽样的处理包括以下三种。

(1) 异常处理

- 旧版本的 Presto 支持 STRATIFY ON 语法, 新版本不再支持, 会抛出异常。
- 抽样比例的表达式中不能包含所抽样表的列名, 否则抛出异常。
- 抽样比例的表达式的计算结果必须为数值型, 否则抛出异常。
- 抽样比例必须大于或等于零, 否则抛出异常。
- 当抽样类型为 BERNOULLI 和 SYSTEM 时, 或者为 POISSONIZED 且指定了 RESCALED 时, 抽样比例必须小于或等于 100, 否则抛出异常。
- 当指定了 RESCALED 但参数 experimental-syntax-enabled 为 false 时, 抛出异常。

(2) 获取并记录抽样表的列描述符

(3) 记录抽样比例

5. TableSubquery

子查询的处理比较简单, 使用 StatementAnalyzer 对子查询的 Query 进行分析, 返回其列描述符即可。

6. QuerySpecification

visitQuerySpecification 方法如下:

```
protected TupleDescriptor visitQuerySpecification(QuerySpecification node,
    AnalysisContext parentContext)
{
    AnalysisContext context = new AnalysisContext(parentContext);

    TupleDescriptor tupleDescriptor = analyzeFrom(node, context);

    node.getWhere().ifPresent(where -> analyzeWhere(node, tupleDescriptor,
        context, where));
}
```

```

List<FieldOrExpression> outputExpressions = analyzeSelect(node,
    tupleDescriptor, context);
List<FieldOrExpression> groupByExpressions = analyzeGroupBy(node,
    tupleDescriptor, context, outputExpressions);
List<FieldOrExpression> orderByExpressions = analyzeOrderBy(node,
    tupleDescriptor, context, outputExpressions);
analyzeHaving(node, tupleDescriptor, context);

analyzeAggregations(node, tupleDescriptor, groupByExpressions,
    outputExpressions, orderByExpressions, context);
analyzeWindowFunctions(node, outputExpressions, orderByExpressions);

TupleDescriptor descriptor = computeOutputDescriptor(node, tupleDescriptor);
analysis.setOutputDescriptor(node, descriptor);
}

```

可见对 QuerySpecification 的处理分为以下几步。

(1) 分析 From 子语句。如果 From 子语句存在，则使用 TupleAnalyzer 进行分析返回列描述符；如果不存在，则返回空的列描述符。

(2) 如果 Where 子语句存在，则分析 Where 子语句

- 如果 Where 子语句中包含聚合操作或窗口函数，则抛出异常。
- 使用表达式分析器 ExpressionAnalyzer 分析 Where 子语句，记录分析后的过滤条件。
- 如果 Where 子语句最后输出的类型不是 BOOLEAN 类型或 NULL 类型，则抛出异常。
- 记录 QuerySpecification 和 Where 条件对应关系。

(3) 分析 Select 子语句

返回结果为 FieldOrExpression 类型的列表，每一个 FieldOrExpression 表示 Select 的一列，其中存储了该列在分析 From 子语句返回的列描述符中的索引下标，或者该列的表达式。循环 Select 子语句的每一列，根据列的类型进行分析。

- 如果列为 * 或 T.* 时，则找到列描述符中所对应的 Field，记录这些 Field 的索引下标，存入 FieldOrExpression 中。
- 如果为单列，则使用表达式分析器 ExpressionAnalyzer 对列表达式进行分析，并存储该列表达式到 FieldOrExpression 中。

(4) 分析 Group By 子语句

返回结果同样为 FieldOrExpression 类型的列表。如果 Group By 子语句存在，则循环其每一列，根据列的类型进行分析。

- 如果列为 Long 类型，则根据该数值查找 Select 子语句返回的 FieldOrExpression 的列表，得到一个 FieldOrExpression。
- 如果列是一个表达式，则使用表达式分析器 ExpressionAnalyzer 对列表达式进行分析，并存储该列表达式到 FieldOrExpression 中。
- 如果列表达式中包含聚合操作或窗口函数，则抛出异常。
- 获取列的类型，如果列类型是不可比较的类型，则抛出异常。

(5) 分析 Order By 子语句

返回结果同样为 `FieldOrExpression` 类型的列表。如果 `Order By` 子语句存在，则进行分析。首先获取 `Select` 子语句中的别名列表，然后循环 `Order By` 子语句的每一列，根据列的类型进行分析。

- 如果列为 `QualifiedNameReference` 类型且其名称无前缀，则从 `Select` 子语句的别名列表中进行查找，如果找不到，则抛出异常。
- 如果列为 `Long` 类型，则根据该数值查找 `Select` 子语句返回的 `FieldOrExpression` 的列表，得到一个 `FieldOrExpression`，获取该列的实际类型，如果列类型是不可排序的，则抛出异常。
- 如果最终得到的 `FieldOrExpression` 包含的是一个表达式，则使用表达式分析器 `ExpressionAnalyzer` 对该表达式进行分析，同样获取列的实际类型，如果该列类型是不可排序的，则抛出异常。
- 如果有 `Select Distinct` 语句且 `Order By` 子语句的列不是 `Select` 列的子集，则抛出异常。

(6) 分析 `Having` 子语句

如果 `Having` 子语句存在，则对其进行分析。使用表达式分析器 `ExpressionAnalyzer` 分析 `Having` 子语句，获取子语句的类型，如果该类型不是 `Boolean` 或 `Unknown` 类型，则抛出异常。

(7) 分析聚合操作

此处分析聚合操作是为了记录聚合操作并检验聚合操作的有效性。

- 首先分析 `Select`、`Order By`、`Having` 子语句，查看其中是否有聚合函数或窗口函数，如果有则获取。
- 如果聚合操作中有 `Distinct` 操作且 `SQL` 语句为近似查询，则抛出异常。
- 分别对 `Select`、`Order By`、`Having` 子语句调用 `verifyAggregations` 方法。
`verifyAggregations` 方法使用 `AggregationAnalyzer` 对以上传入的子语句进行分析。
- 如果子语句中的某些列没有应用聚合函数或窗口函数，而且不在 `Group By` 子语句中，则抛出异常。

(8) 分析窗口函数

对窗口函数的分析分为以下几步。

- 解析出 `Select`、`Order By` 子语句中所有的窗口函数，并检验窗口函数的使用是否合法。
- 分析窗口函数中是否嵌套了其他的窗口函数，如果有则抛出异常。
- 窗口函数中目前不支持 `Distinct`，因此，如果存在这种情况，则抛出异常。
- 如果窗口函数有 `WindowFrame` 属性，则检验 `WindowFrame` 的有效性。
- 获取每个窗口函数的参数，并与系统中注册的窗口函数进行对比，如果找不到对应的窗口函数，则抛出异常。

(9) 获取输出的列描述符

根据 `From` 子语句获取的列描述符与 `Select` 子语句中的列，计算出最终需要输出的列描述符。

7. Union

visitUnion 方法对于 Union 的处理分为以下几步。

- 检验 Union 的语句个数是否大于或等于 2，若小于 2，则抛出异常。
- TupleAnalyzer 分析 Union 的第一个子语句，获取其列描述符 outputDescriptor。
- 分析 Union 剩下的子语句，以 outputDescriptor 为基准与其进行对比，如果列数不同或者存在列的类型不匹配的情况，则抛出异常。

8. Intersect

目前不支持该类型语句的处理。

9. Except

目前不支持该类型语句的处理。

10. Join

visitJoin 对 Join 语句的处理主要分为以下几步。

- 获取 Join 语句的连接条件，若该连接为自然连接，则抛出异常。
- 对 Join 语句左右两侧的 relation 分别进行分析，并获取列描述符。
- 如果 Join 两侧的列描述符有重复的列名或列别名，则抛出异常。
- 合并 Join 两侧的列描述符，作为 Join 语句输出的列描述符。
- 如果 Join 类型为 Cross Join 或 Implicit Join，则返回以上的列描述符，否则继续。
- 如果 Join 连接条件为 Join Using，则针对 Using 中的每一列：
 - 分析 Join 两侧 Relation 的对应列的表达式。
 - 如果两侧对应列的类型不匹配，则添加类型转换。
 - 最后根据这些列拼接成普通的相等比较的 Join On 连接语句并记录。
- 如果 Join 连接条件为 Join On，则：
 - 使用表达式分析器 ExpressionAnalyzer 分析 Join On 的连接条件。
 - 确保连接条件中没有聚合函数和窗口函数。
 - 对连接条件语句进行标准化，主要是一些时间函数以及 If 语句转为 Case When 这样的处理。
 - 使用表达式解释器 ExpressionInterpreter 对上一步的语句进行进一步的优化。
 - 如果连接条件语句最终优化成了一个布尔值，则根据布尔值拼接 0=0 或 0=1 这样的表达式，替换原语句。
 - 如果最终的连接条件语句不是布尔类型的表达式，则抛出异常。
 - 由于以上的语句优化过程可能破坏了一些表达式的类型转换，因此使用表达式分析器 ExpressionAnalyzer 重新分析优化后的语句。
 - 针对每一组连接条件，区分开其中的表达式是属于 Join 左侧还是右侧的 Relation，并分别对这些表达式进行分析。
- 如果 Join 连接条件既不是 Join Using 也不是 Join On，则抛出异常。

- 记录以上合并后的列描述符并返回。

11. Values

visitValues 对 Values 语句进行分析分为以下几步。

- 获取每一行每一个元素的类型。
- 获取每一列中所有元素的共同的超级类型，例如 bigint 与 double 共有的超级类型为 double。
- 如果存在元素的类型不是其所在列的超级类型，则为其添加类型转换。
- 根据每一列的超级类型构造列描述符并返回。

5.4.3 表达式分析

ExpressionAnalyzer 类对 SQL 语句中的表达式进行分析，主要功能如下。

- (1) 获取表达式的类型。
- (2) 获取需要进行类型转换的表达式及其转换的目的类型。
- (3) 获取表达式中存在的函数信息。
- (4) 获取表达式中所有合法的列名及对应列的编号。
- (5) 获取表达式中 In 语句中的子查询。

5.5 执行计划生成

LogicalPlanner 类会根据以上针对 SQL 语句分析所得的结果，生成逻辑执行计划。

5.5.1 执行计划节点

在讲解执行计划生成之前，首先介绍一下执行计划树中的节点类型。

(1) AggregationNode

AggregationNode 是用于聚合操作的节点，聚合的类型有 FINAL、PARTIAL、SINGLE 三种，分别表示最终聚合、局部聚合和单点聚合，其中执行计划在优化之前，聚合的类型都是单点聚合，在执行计划优化器中会对其进行拆分成局部聚合和最终聚合。

(2) DeleteNode

DeleteNode 是用于 DELETE 操作的节点。

(3) DistinctLimitNode

DistinctLimitNode 是用于处理以下类型的 SQL 语句的节点。

```
SELECT DISTINCT ... FROM ... LIMIT ...
```

(4) ExchangeNode

ExchangeNode 是用于在执行计划中不同 Stage 之间交换数据的节点，出现在逻辑执行计划中。

(5) FilterNode

FilterNode 是用于进行过滤操作的节点。

(6) IndexJoinNode

IndexJoinNode 是用于对 Index Join 操作的节点。

(7) IndexSourceNode

IndexSourceNode 是与 Index Join 配合使用的执行数据源读取操作的节点。

(8) JoinNode

JoinNode 是执行 Join 操作的节点。

(9) LimitNode

LimitNode 是执行 Limit 操作的节点。

(10) MarkDistinctNode

MarkDistinctNode 是用于处理以下类型的 SQL 语句的节点。

```
SELECT COUNT(DISTINCT ...) FROM ...
```

(11) OutputNode

OutputNode 是用于输出最终结果的节点。

(12) ProjectNode

ProjectNode 是用于进行列映射的节点，用于将 ProjectNode 下层节点输出的列映射到 ProjectNode 上层节点输入的列，例如：

```
SELECT l_orderkey+1 FROM LINEITEM;
```

ProjectNode 会将其下层 TableScanNode 输出的 l_orderkey 作为输入，映射为 l_orderkey+1 并传递给上层节点。

(13) RemoteSourceNode

RemoteSourceNode 类似 ExchangeNode，用于分布式执行计划中不同的 Stage 之间交换数据，出现在分布式执行计划中。

(14) RowNumberNode

RowNumberNode 用于处理窗口函数 row_number。

(15) SampleNode

SampleNode 用于处理抽样函数。

(16) SemiJoinNode

SemiJoinNode 用于处理执行计划生成过程中产生的 SemiJoin。

(17) SortNode

SortNode 用于排序操作。

(18) TableCommitNode

TableCommitNode 用于对 Create Table As Select 语句、Insert 语句、Delete 语句的操作进行 Commit。

(19) TableScanNode

TableScanNode 用于读取表的数据。

(20) TableWriterNode

TableWriterNode 用于向目的表写入数据。

(21) TopNNode

TopNNode 用于取数据排序后的前 N 条结果，使用效率更高的 TopN 算法，而不是对所有数据进行全局排序再取前 N 条，TopN 问题与算法不再具体详述，请自行查找资料，如：

```
SELECT l_orderkey FROM LINEITEM ORDER BY l_orderkey limit 10;
```

该 SQL 会取最小的 10 条 l_orderkey，Presto 会选取 TopNNode 而不是 SortNode 来进行排序取最小的 10 条 l_orderkey。

(22) TopNRowNumberNode

TopNRowNumberNode 用于处理窗口函数 row_number 中排序取前 N 条结果，使用效率更高的 TopN 算法，如：

```
SELECT x FROM (SELECT x, row_number() OVER (ORDER BY x) rn FROM ...) WHERE rn < 11;
```

该 SQL 中的子查询用于取 x，以及按 x 升序排列的行号，最后取出行号小于 11 的 x。Presto 会选取 TopNRowNumberNode 来选取行号小于 11 的 x，即最小的 10 条 x。

(23) UnionNode

UnionNode 用于处理 Union 操作。

(24) UnnestNode

UnnestNode 用于处理 Unnest 操作。

(25) ValuesNode

ValuesNode 用于处理 Values 语句。

(26) WindowNode

WindowNode 用于处理窗口函数。

5.5.2 SQL 执行计划

LogicalPlanner 负责整个 SQL 语句执行计划的生成，根据 SQL 语句的类型生成不同的执行计划，然后针对生成的执行计划，分别使用已注册的执行计划优化器进行优化，其主要代码如下：

```
public Plan plan(Analysis analysis)
{
    RelationPlan plan;
    if (analysis.getCreateTableDestination().isPresent()) {
        plan = createTableCreationPlan(analysis);
    }
    else if (analysis.getInsertTarget().isPresent()) {
        plan = createInsertPlan(analysis);
    }
}
```

```
    }
    else if (analysis.getDelete().isPresent()) {
        plan = createDeletePlan(analysis);
    }
    else {
        plan = createRelationPlan(analysis);
    }

    PlanNode root = createOutputPlan(plan, analysis);

    // 确保生成的执行计划有效
    PlanSanityChecker.validate(root);

    // 执行计划优化器
    for (PlanOptimizer optimizer : planOptimizers) {
        root = optimizer.optimize(root, session, symbolAllocator.getTypes(),
            symbolAllocator, idAllocator);
        checkNotNull(root, "%s returned a null plan", optimizer.getClass().
getName());
    }

    // 再次确保生成的执行计划有效
    PlanSanityChecker.validate(root);

    return new Plan(root, symbolAllocator);
}
```

1. TableWriter Plan

Create Table As Select 语句和 Insert 语句都会生成 TableWriterPlan, 其所生成的执行计划树结构如图 5-14 所示。

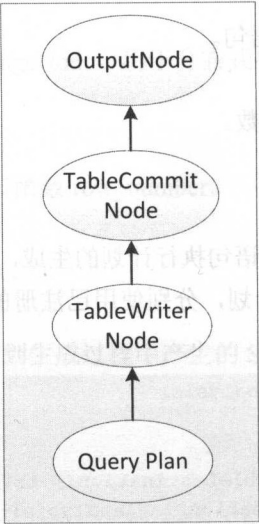


图 5-14

其中 QueryPlan 是指 Create Table As Select 语句或 Insert 语句后面的查询语句生成的执行计划树。

在 TableWriterNode 和 OutputNode 之间添加 TableCommitNode 可以防止数据写入失败导致的中间状态，当确保数据写入成功之后再进行 Commit 操作。

2. DeletePlan

Delete 语句生成 DeletePlan，其执行计划树结构如图 5-15 所示。

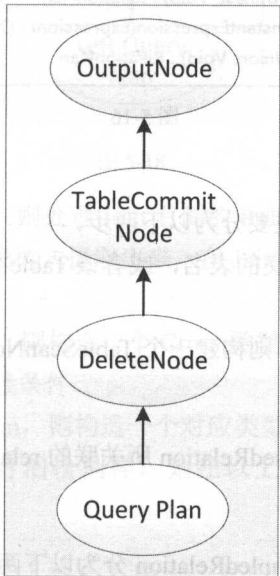


图 5-15

同样，图 5-15 中的 QueryPlan 为 Delete 语句之后的查询语句所生成的执行计划树。

3. QueryPlan

所有 Relation 类型的 SQL 语句都会生成 QueryPlan，由下一节中的 RelationPlanner 分析并生成查询执行计划。

5.5.3 Relation 执行计划

RelationPlanner 用于针对 Relation 类型的 SQL 语句生成执行计划，其类视图如图 5-16 所示。

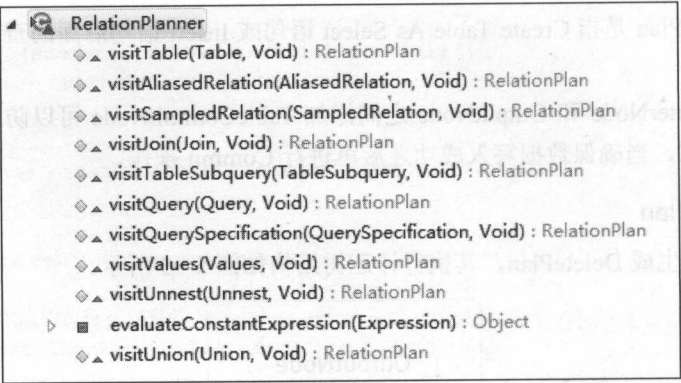


图 5-16

1. Table

visitTable 对 Table 进行分析主要分为以下两步。

- 如果该 Table 是 With 所定义的表名，或者该 Table 实际是一个 View，则处理其所关联的查询生成执行计划。
- 如果该 Table 是普通的表，则构建一个 TableScanNode。

2. AliasedRelation

visitAliasedRelation 处理 AliasedRelation 所关联的 relation，并生成执行计划。

3. SampledRelation

visitSampledRelation 处理 SampledRelation 分为以下两步（见图 5-17）。

- 处理其关联的 Relation，生成执行计划树。
- 构建一个 SampleNode，添加到以上的执行计划树之上。

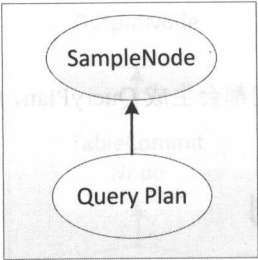


图 5-17

4. Join

visitJoin 处理 Join 分为以下几步。

- 处理 Join 左侧的 Relation，生成左侧执行计划树。
- 如果 Join 右侧是 Unnest 且 Join 类型为 Cross Join 或 Implicit Join，则根据 Unnest 构造一个 UnnestNode 以及一个 ProjectNode，添加到左侧执行计划树之上并返回，如图 5-18 所示。

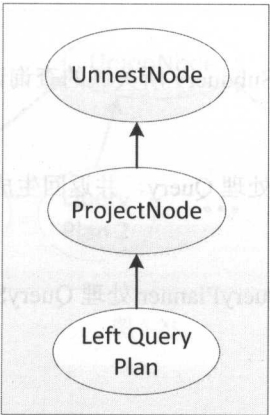


图 5-18

- 如果 Join 右侧不是 Unnest，则处理 Join 右侧的 Relation，生成右侧执行计划树。
- 将 Join 连接条件中的列或表达式提取出来，在 Join 两侧的执行计划树上分别添加一个 ProjectNode。
- 如果 Join 类型是 Inner Join，则构造一个 Cross 类型的 JoinNode，并在之上添加一个 FilterNode，过滤条件为连接条件。
- 如果 Join 类型不是 Inner Join，则构造一个对应类型的 JoinNode。
- 如果 Join 左侧或右侧含有泊松抽样，则在以上的执行计划树之上再添加一个 ProjectNode。

示意图如图 5-19 所示。

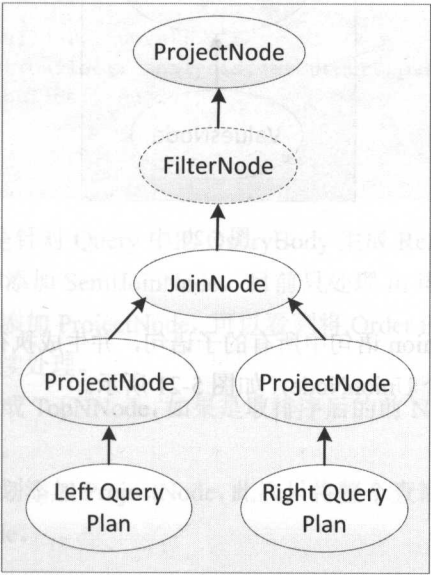


图 5-19

其中虚线的 FilterNode 表示该节点根据具体 SQL 语句可能存在，也可能不存在。

5. TableSubquery

visitTableSubquery 处理 TableSubquery 所关联的查询语句并返回生成的执行计划。

6. Query

visitQuery 使用 QueryPlanner 处理 Query，并返回生成的执行计划。

7. QuerySpecification

visitQuerySpecification 使用 QueryPlanner 处理 QuerySpecification，并返回生成的执行计划。

8. Values

visitValues 针对 Values 语句，获取其中每一项的值，并构造 ValuesNode。

9. Unnest

Cross Join Unnest 在 visitJoin 中已经处理过了，visitUnnest 只会处理常量类型的 Unnest 语句：

```
SELECT * FROM UNNEST(ARRAY[1, 2, 3]);  
SELECT a, b FROM UNNEST(MAP(ARRAY[1,2], ARRAY['cat', 'dog'])) t(a, b);
```

首先根据 Unnest 的常量构造 ValuesNode，然后在 ValuesNode 之上再构造一个 UnnestNode，如图 5-20 所示。

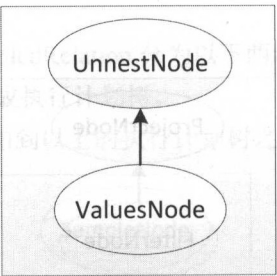


图 5-20

10. Union

visitUnion 分别处理 Union 语句中所有的子语句，并生成执行计划树，最后在这些执行计划树的根节点上构造一个 UnionNode，如图 5-21 所示。

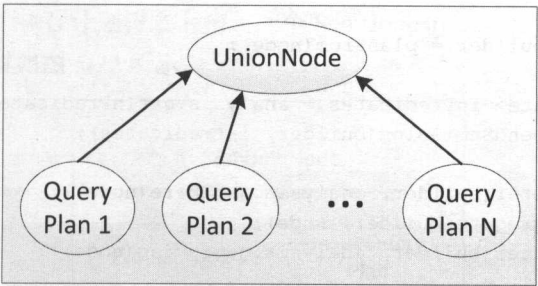


图 5-21

5.5.4 Query 执行计划

QueryPlanner 用于处理 Query 和 QuerySpecification。

1. Query

visitQuery 方法处理 Query:

```
protected PlanBuilder visitQuery(Query query, Void context)
{
    PlanBuilder builder = planQueryBody(query);
    Set<InPredicate> inPredicates = analysis.getInPredicates(query);
    builder = appendSemiJoins(builder, inPredicates);

    List<FieldOrExpression> orderBy = analysis.getOrderByExpressions(query);
    List<FieldOrExpression> outputs = analysis.getOutputExpressions(query);
    builder = project(builder, Iterables.concat(orderBy, outputs));

    builder = sort(builder, query);
    builder = project(builder, analysis.getOutputExpressions(query));
    builder = limit(builder, query);

    return builder;
}
```

- planQueryBody 就是针对 Query 中的 QueryBody 生成 Relation 执行计划。
- appendSemiJoins 即添加 SemiJoinNode，目前只处理 In 语句中的子查询。
- project 为执行计划添加 ProjectNode，可以看到将 Order By 的列和输出列都映射到了上层节点，以便后续处理。
- sort 添加 SortNode 或 TopNNode，如果是取排序后的前 N 条结果，则添加 TopNNode，否则添加 SortNode。
- project 再为执行计划添加 ProjectNode，此次只将整个查询的输出列映射到上层节点。
- limit 添加 LimitNode。

2. QuerySpecification

```
protected PlanBuilder visitQuerySpecification(QuerySpecification node, Void context)
```

```

{
    PlanBuilder builder = planFrom(node);

    Set<InPredicate> inPredicates = analysis.getInPredicates(node);
    builder = appendSemiJoins(builder, inPredicates);

    builder = filter(builder, analysis.getWhere(node));
    builder = aggregate(builder, node);
    builder = filter(builder, analysis.getHaving(node));

    builder = window(builder, node);

    List<FieldOrExpression> orderBy = analysis.getOrderByExpressions(node);
    List<FieldOrExpression> outputs = analysis.getOutputExpressions(node);
    builder = project(builder, Iterables.concat(orderBy, outputs));

    builder = distinct(builder, node, outputs, orderBy);
    builder = sort(builder, node);
    builder = project(builder, analysis.getOutputExpressions(node));
    builder = limit(builder, node);

    return builder;
}

```

- planFrom 即处理 From 语句，如果不存在 From 语句，则表明 SQL 是 Select 1 这样的简单语句，直接构造 ValuesNode 返回，否则使用 RelationPlanner 处理 FROM 语句生成 Relation 执行计划。
- appendSemiJoins 处理同上。
- filter 为 Where 条件添加 FilterNode。
- aggregate 处理聚合操作。
- filter 再为 Having 子语句添加 FilterNode。
- window 处理窗口函数。
- project 处理同上。
- distinct 处理 Distinct 语句，添加 AggregationNode。
- sort 处理同上。
- project 处理同上。
- limit 处理同上。

在以上的处理中，SemiJoin、Aggregate 操作和 Window 函数处理比较复杂，以下分别进行讲解。

3. SemiJoin

SemiJoin 目前处理针对 In 语句中的子查询，处理步骤如下。

- 为 In 语句左侧生成的执行计划树添加 ProjectNode，将 In 语句中的子语句的列也作为输出传递给上层节点。
- 使用 RelationPlanner 处理 In 语中的子语句，生成执行计划树。

- 最后在以上两个执行计划树之上构造一个 SemiJoinNode。
- 最后的执行计划树如图 5-22 所示。

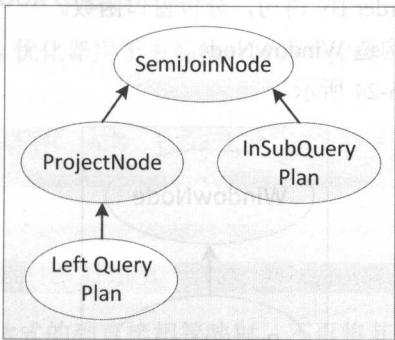


图 5-22

4. Aggregate 操作

Aggregate 操作的处理分为如下步骤。

- 获取聚合函数的参数以及 Group By 语句的表达式，依此构造 ProjectNode。
- 对聚合参数与 Group By 语句进行分析。
- 如果语句中含有 Distinct，则构造 MarkDistinctNode。
- 依据之前的分析结果构造 AggregationNode。

最后的执行计划树如图 5-23 所示。

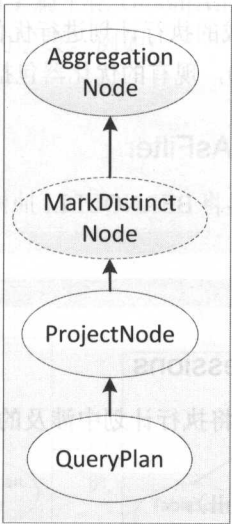


图 5-23

虚线的 MarkDistinctNode 表示根据具体 SQL 语句该节点可能存在也可能不存在。

5. Window 函数

窗口函数的处理分为如下步骤。

- 获取窗口函数的参数以及其中的 Partition By 和 Order By 语句的表达式，依此构造 ProjectNode。
 - 分析 Partition By 和 Order By 语句，分析窗口函数。
 - 按照以上的分析结果构造 WindowNode。
- 最后的执行计划树如图 5-24 所示。

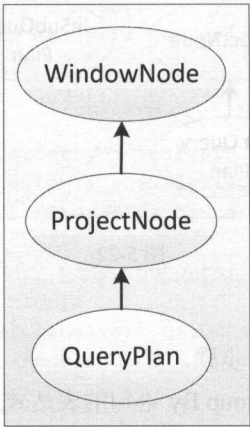


图 5-24

5.6 执行计划优化

生成执行计划之后，会对所生成的执行计划进行优化，目前 Presto 只支持基于规则的优化器（RBO Rule-Based Optimizer），现有的优化器包括如下几种。

5.6.1 ImplementSampleAsFilter

ImplementSampleAsFilter 优化器将 BERNOULLI 抽样的 SampleNode 改写为 FilterNode，FilterNode 的过滤条件为：

```
rand() < SampleRatio
```

5.6.2 CanonicalizeExpressions

CanonicalizeExpressions 优化器将执行计划中涉及的表达式进行标准化，标准化的主要工作有：

- Is Not Null 改写为 Not (Is Null)。
- If 语句改写为 Case When 语句。
- 处理时间函数。

5.6.3 SimplifyExpressions

SimplifyExpressions 优化器对执行计划中涉及的表达式进行简化和优化处理，具体可查

看 ExpressionInterpreter。

5.6.4 UnaliasSymbolReferences

UnaliasSymbolReferences 优化器用于去除执行计划中 ProjectNode 中的无意义映射，例如对于以下的执行计划

```
Output[$0, $1] -> Project[$0 := $2, $1 := $3 * 100] -> Aggregate[$2, $3 := sum($4)]
-> ...
```

改写为

```
Output[$2, $1] -> Project[$2, $1 := $3 * 100] -> Aggregate[$2, $3 := sum($4)] -> ...
```

可以看到，对于不带表达式的列直接相等映射，不再将其映射为新的列名，而是将列直接映射到上层节点中。

5.6.5 PruneRedundantProjections

PruneRedundantProjections 优化器用于去除多余的 ProjectNode，如果 ProjectNode 中都是如下的列直接相等映射，则去除该 ProjectNode：

```
Project[$0 := $0, $1 := $1]
```

5.6.6 SetFlatteningOptimizer

SetFlatteningOptimizer 优化器用于扁平化 Union[All]语句的执行计划树，即合并能够合并的 Union 节点，减少执行计划树的层数。例如：

```
SELECT * FROM
(SELECT * FROM lineitem
UNION
SELECT * FROM lineitem)
UNION
SELECT * FROM lineitem
```

该优化器效果如图 5-25 所示。

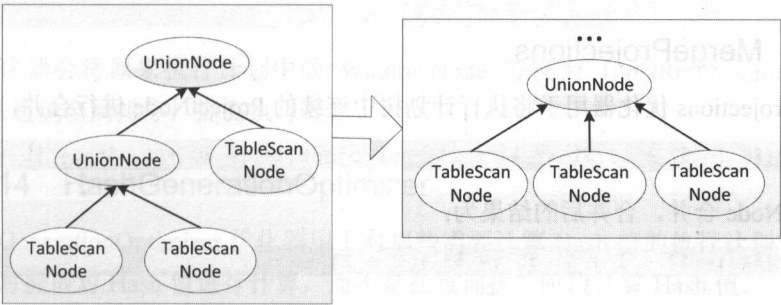


图 5-25

可见两个 UnionNode 进行了合并，执行计划树减少了一层。

5.6.7 LimitPushDown

LimitPushDown 优化器用于将 limit 条件进行下推，从而减少下层节点所处理的数据量，提高执行效率。

- 如果 limit 值为 0，则不再进行下推，直接返回一个值为空的 ValuesNode。
- 当 limit 条件下推到 AggregationNode 时，如果是 Distinct 聚合，则将 AggregationNode 替换为 DistinctLimitNode。
- 当 limit 条件下推到 TopNNode 时，根据 limit 的值和 TopNNode 中 N 的值，取其最小值构造一个新的 TopNNode。
- 当 limit 条件下推到 SortNode 时，将 SortNode 替换为 TopNNode。
- 当 limit 条件下推到 UnionNode 时，在 UnionNode 的每一个子节点之上添加一个 LimitNode。

5.6.8 PredicatePushDown

PredicatePushDown 优化器用于将过滤条件进行下推，同样可以减少下层节点所处理的数据量，提高执行效率。

- 当经过 ProjectNode 时，仅下推确定性的过滤条件，如果包含不确定性的过滤条件如 rand 函数，则根据不确定性的过滤条件添加一个 FilterNode。
- 当经过 UnionNode 时，将过滤条件下推到 Union 的所有子查询中。
- 对于 JoinNode，由于 Join 操作对应的是两张表，如果过滤条件是针对这两张表的，则将过滤条件下推到读取对应的表上，并且如果过滤条件的列与连接条件中的列相同时，则将过滤条件同时下推到 Join 的左右两侧的表上。如：

```
SELECT * FROM lineitem JOIN orders ON l_orderkey = o_orderkey where l_orderkey = 1;
```

过滤条件 `l_orderkey = 1` 会被下推到读取表 `lineitem` 之上，过滤条件 `o_orderkey=1` 会被下推到读取表 `orders` 之上。

5.6.9 MergeProjections

MergeProjections 优化器用于将执行计划树中连续的 ProjectNode 进行合并，例如将

```
Project[$0 := $2 + 1, $1 := $3 + 2] -> Project[$4 := $0 + 1, $5 := $1 + 2] -> ...
```

会将 ProjectNode 合并，合并后的结果为：

```
Project[$4 := $2 + 1 + 1, $5 := $3 + 2 + 2] -> ...
```

5.6.10 ProjectionPushDown

ProjectionPushDown 优化器将 UnionNode 之上的 ProjectNode 下推到 UnionNode 之下，

有利于减少 UnionNode 所处理的列数，从而减少数据处理量，提高处理效率，效果如图 5-26 所示。

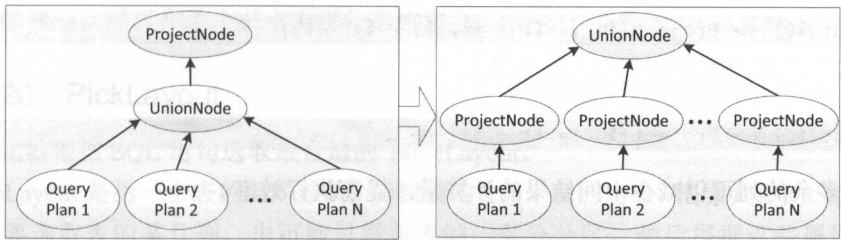


图 5-26

5.6.11 IndexJoinOptimizer

IndexJoinOptimizer 优化器用于将 Join 优化为 IndexJoin，获取做 Join 操作的表的 Index，并依此进行 Join 操作。

5.6.12 CountConstantOptimizer

CountConstantOptimizer 优化器用于将 count(constant) 改写为 count(*), 例如将

```
SELECT COUNT (42) FROM ORDERS
```

改写为

```
SELECT COUNT (*) FROM ORDERS
```

在 Presto 中，count(constant)与 count(*)是等效的，但是 count(*)取表的行数更容易根据不同的数据源进行优化。

5.6.13 WindowFilterPushDown

WindowFilterPushDown 优化器用于处理窗口函数 row_number 中排序取前 N 条结果，如以下代码所示。

```
SELECT x FROM (SELECT x, row_number() OVER (ORDER BY x) rn FROM ...) WHERE rn < 10;
```

该优化器会将原来执行计划中的 WindowNode 替换为 TopNRowNumberNode，应用 TopN 算法避免全局排序，提高执行效率。

5.6.14 HashGenerationOptimizer

HashGenerationOptimizer 优化器用于对那些需要计算 Hash 值的执行计划节点，在执行计划阶段时提前对 Hash 值进行计算，而不是在查询执行阶段计算 Hash 值。

5.6.15 PruneUnreferencedOutputs

PruneUnreferencedOutputs 优化器用于去除 ProjectNode 中不在最终输出中的列，例如将

```
Output[$0] -> Project[$0 := $1 + $2, $3 = $4 / $5] -> ...
```

改写为

```
Output[$0] -> Project[$0 := $1 + $2] -> ...
```

去除多余的列可以减少中间结果的计算量，提高执行效率。

5.6.16 MetadataQueryOptimizer

该优化器用于优化只对表的分区字段进行的聚合操作，将其改写为针对表元数据的查询，省去了读取原始表的操作。

例如以下的语句：

```
SELECT min(key), max(key) FROM t
SELECT DISTINCT key FROM t
SELECT count(DISTINCT key) FROM t
SELECT count(DISTINCT key + 5) FROM t
SELECT count(DISTINCT key) FROM (SELECT key FROM t ORDER BY 1 LIMIT 10);
SELECT key1, count(DISTINCT key2) FROM t GROUP BY 1
```

字段 key、key1、key2 都是分区字段，这些语句经过优化后都会被改写为针对表元数据的查询。

5.6.17 SingleDistinctOptimizer

SingleDistinctOptimizer 优化器用于将单一的 Distinct 聚合优化为 Group By，例如将

```
SELECT x, COUNT(DISTINCT y) FROM foo GROUP BY x
```

优化为

```
SELECT x, count(y)
FROM
  (SELECT x, y
   FROM foo
   GROUP BY x, y)
GROUP BY x
```

该优化器可以尽量将数据进行本地聚合，从而减少后续的数据传输，提升执行效率。

5.6.18 BeginTableWrite

确切地说，BeginTableWrite 并不是一个优化器，其作用是根据 SQL 语句类型调用 beginCreate/beginInsert，然后构造 CreateHandle/InsertHandle，用于 TableWriterNode 后续操作。

5.6.19 AddExchanges

该优化器根据逻辑执行计划生成分布式执行计划。例如针对聚合操作，先在多个节点上进行局部聚合，最后在单个节点上进行最终聚合。

5.6.20 PickLayout

该优化器根据 SQL 语句选取最合适的 TableLayout。

TableLayout 是指一个表的组织结构，例如表的分区、表的列等。根据具体的 SQL 语句，可能只需要选取表的某几列，也可能只需要表的一部分分区，最后根据这些属性构造的 TableLayout 就是最适合当前 SQL 语句的 TableLayout。

5.7 执行计划分段

经过执行计划生成与执行计划优化之后，最后对执行计划进行分段：

```
SubPlan subplan = new PlanFragmenter().createSubPlans(plan);
```

Presto 根据执行计划的操作分为如下几个阶段。

5.7.1 Source

顾名思义，Source 阶段是从数据源的表中读取数据的阶段，一般包括 TableScanNode 和 ProjectNode，以及可能存在的 FilterNode 等。

5.7.2 Fixed

Fixed 阶段位于 Source 阶段之后，该阶段将 Source 阶段读取的数据分散到多个节点上进行处理，主要处理的操作有局部聚合、局部 Join、局部数据写入表等。

5.7.3 Single

Single 阶段位于 Fixed 阶段之后，只在单个节点上执行，用于汇总所有的处理结果，例如针对局部聚合的数据进行最终聚合，并将结果传输给 Coordinator。

另外全局排序的操作也是在 Single 阶段进行。

5.7.4 Coordinator_only

Coordinator_only 阶段只在 Coordinator 上进行，对 Insert 和 Create table 操作进行 Commit 的 TableCommitNode 属于 Coordinator_only 阶段。

5.8 示例

以下示例给出比较常见的两种 SQL 语句的查询执行计划。

5.8.1 Count 执行计划

Count 语句的执行计划如下：

```
presto:orc> explain (type distributed) select count(*) from lineitem;
Query Plan
-----
Fragment 0 [SINGLE]
  Output layout: [count]
  - Output[_col0] => [count:bigint]
    _col0 := count
  - Aggregate(FINAL) => [count:bigint]
    count := "count"("count_3")
  - RemoteSource[1] => [count_3:bigint]

Fragment 1 [SOURCE]
  Output layout: [count_3]
  - Aggregate(PARTIAL) => [count_3:bigint]
    count_3 := "count"(*)
  - TableScan[hive:hive:orc:lineitem, originalConstraint = true] => []
  LAYOUT: hive
```

可见其执行计划分为两个阶段：Source 和 Single，在 Source 阶段同时进行了局部聚合，在 Single 阶段进行最终聚合。

执行计划如图 5-27 所示。

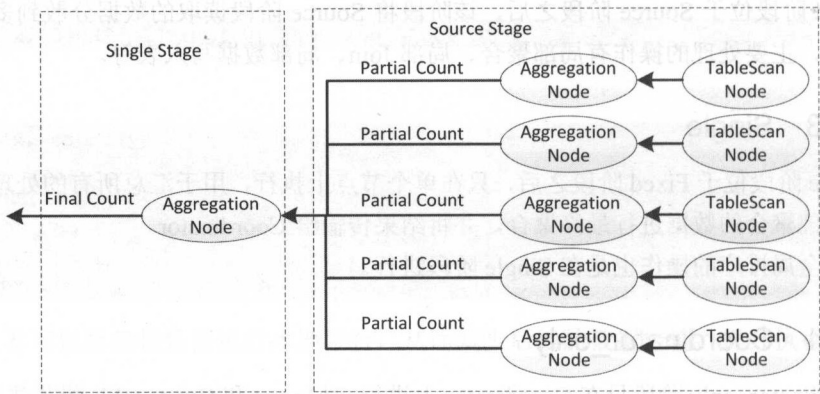


图 5-27

5.8.2 Join 执行计划

Join 语句的执行计划如下：

```
presto:orc> explain (type distributed) SELECT l_partkey FROM lineitem JOIN orders
```

```

ON l_orderkey = o_orderkey where l_orderkey = 1;
Query Plan
-----
Fragment 0 [SINGLE]
  Output layout: [l_partkey]
  - Output[l_partkey] => [l_partkey:bigint]
    - RemoteSource[1] => [l_partkey:bigint]

Fragment 1 [FIXED]
  Output layout: [l_partkey]
  - Project => [l_partkey:bigint]
    - InnerJoin[("expr_8" = "expr_9")] => [l_partkey:bigint, $hashvalue:
      bigint, expr_8:bigint, $hashvalue_7:bigint, expr_9:bigint]
      - Project => [l_partkey:bigint, $hashvalue:bigint, expr_8:bigint]
        expr_8 := 0
      - RemoteSource[2] => [l_partkey:bigint, expr:bigint, $hashvalue:
        bigint]
      - Project => [$hashvalue_7:bigint, expr_9:bigint]
        expr_9 := 0
      - RemoteSource[3] => [expr_6:bigint, $hashvalue_7:bigint]

Fragment 2 [SOURCE]
  Output layout: [l_partkey, expr, $hashvalue]
  Output partitioning: [expr]
  - Project => [l_partkey:bigint, expr:bigint, $hashvalue:bigint]
    expr := 0
    $hashvalue := "combine_hash"(0, COALESCE("$operator$hash_code"(0), 0))
  - Filter[("l_orderkey" = 1)] => [l_orderkey:bigint, l_partkey:bigint]
    - TableScan[hive:hive:orc:lineitem, originalConstraint = (1 =
      "l_orderkey")] => [l_orderkey:bigint, l_partkey:bigint]
      LAYOUT: hive
      l_orderkey := HiveColumnHandle{clientId=hive, name=l_orderkey,
        ordinalPosition=0, hiveType=bigint, hiveColumnIndex=0, partitionKey=false}
      l_partkey := HiveColumnHandle{clientId=hive, name=l_partkey,
        ordinalPosition=1, hiveType=bigint, hiveColumnIndex=1, partitionKey=false}

Fragment 3 [SOURCE]
  Output layout: [expr_6, $hashvalue_7]
  Output partitioning: [expr_6]
  - Project => [expr_6:bigint, $hashvalue_7:bigint]
    expr_6 := 0
    $hashvalue_7 := "combine_hash"(0, COALESCE("$operator$hash_code"(0),
      0))
  - Filter[("o_orderkey" = 1)] => [o_orderkey:bigint]
    - TableScan[hive:hive:orc:orders, originalConstraint = (1 =
      "o_orderkey")] => [o_orderkey:bigint]
      LAYOUT: hive
      o_orderkey := HiveColumnHandle{clientId=hive, name=o_orderkey,
        ordinalPosition=0, hiveType=bigint, hiveColumnIndex=0, partitionKey=false}

```

执行计划如图 5-28 所示。

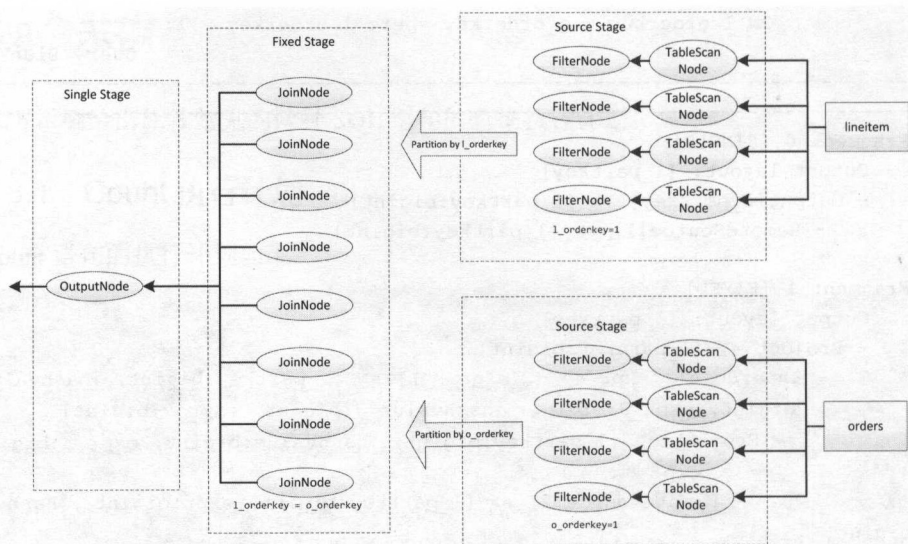


图 5-28

5.9 小结

本章详细描述了一个 SQL 的执行计划生成的过程，对词法分析、语法分析、语义分析、执行计划生成、执行计划优化、执行计划分段等进行了深入的讲解，最后给出了几个常用 SQL 的执行计划的示例。通过本章的学习，读者能够对查询执行计划中涉及的基本概念有所认识，并且对查询执行计划的生成以及优化有比较深入的了解。

第 6 章

查询调度

从第 5 章我们了解到, Presto 根据生成的逻辑执行计划将其拆分成多个且有层级关系的 Stage, Presto 的各个 Stage 都可以理解为将整个查询的执行计划分成了若干段, 实际就是将整个 SQL 处理过程拆分成多个具有各自功能的执行阶段。每个执行阶段都会被进一步分解为若干个 Task, Presto 将查询请求解析成各个执行阶段后, 便会将各个阶段分配到各个计算节点中执行, 这个分配的过程实际是基于 Stage 进行的, 每个 Stage 的调度过程是基于 Split 分配 Worker Node 的过程, 不同的 Stage 有不同的调度策略。

6.1 生成调度执行器

Stage 层级关系的形成是通过查询执行计划演变而来的, 对于每一个 Stage, Presto 都会生成一个与其对应的 SqlStageExecution 实例, SqlStageExecution 承载了 Stage 的启动和任务的调度, 也可称之为整个查询的调度执行器。SQL 语句经过 `com.facebook.presto.execution.SqlQueryExecution.analyzeQuery` 方法后, 获取 SQL 的逻辑执行计划 `com.facebook.presto.sql.planner.SubPlan`。SubPlan 有两个成员变量: PlanFragment fragment 和 `List<SubPlan> children`, 其中 children 表示子查询计划, 通过递归, child 可遍历整个查询计划。整个查询的调度过程如图 6-1 所示。以下过程为通过 SubPlan 来调度起整个 SQL 所有的 Stage。

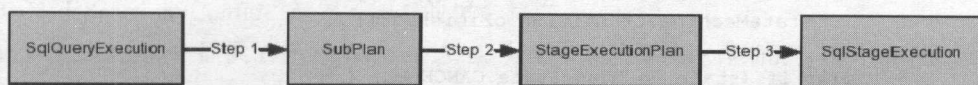


图 6-1

- Step 1 由 `SqlQueryExecution.analyzeQuery` 方法生成 subPlan。
- Step 2 根据获取的 subPlan 信息, 在 `SqlQueryExecution.planDistribution` 方法调用了 `DistributedExecutionPlanner.plan` 获得 Stage 的执行计划: StageExecutionPlan。代码具

体内容如下：

```
public StageExecutionPlan plan(SubPlan root)
{
    PlanFragment currentFragment = root.getFragment();
    .....
    for (SubPlan childPlan : root.getChildren()) {
        //递归的获取了所有的子计划
        dependencies.add(plan(childPlan));
    }

    return new StageExecutionPlan(currentFragment,
        splits,
        dependencies.build()
    );
}
```

StageExecutionPlan 中维护了一个子 Stage 执行计划集合: List<StageExecutionPlan> subStages, 其中最上层的 Stage 为 outputStage, 用于最终结果输出。

- Step 3 由最上层的结果输出阶段的执行计划: StageExecutionPlan outputStage, 以及其他一些参数共同组成了 SqlStageExecution 所需的成员变量, 其过程如下:

```
private void planDistribution(SubPlan subplan)
{
    .....
    //创建 stage execution 对象

    SqlStageExecution outputStage = new SqlStageExecution(stateMachine.
getQueryId(),
        locationFactory,
        outputStageExecutionPlan,
        nodeScheduler,
        remoteTaskFactory,
        stateMachine.getSession(),
        scheduleSplitBatchSize,
        initialHashPartitions,
        queryExecutor,
        nodeTaskMap,
        ROOT_OUTPUT_BUFFERS);
    // 注册监听器
    outputStage.addStateChangeListener(state -> {
        if (state == StageState.FINISHED) {
            stateMachine.transitionToFinished();
        }
        else if (state == StageState.CANCELED) {
            stateMachine.transitionToFailed(new PrestoException(USER_CANCELED,
"Query was canceled"));
        }
    });
    .....
    // 设置全局的 outputStage
    this.outputStage.set(outputStage);
}
```

在 `SqlStageExecution` 本身的构造方法中,通过递归的方式构造子 `Stage` 对象,代码如下:

```
ImmutableMap.Builder<PlanFragmentId, SqlStageExecution> subStages = ImmutableMap.  
builder();  
for (StageExecutionPlan subStagePlan : plan.getSubStages()) {  
    PlanFragmentId subStageFragmentId = subStagePlan.getFragment().getId();  
    SqlStageExecution subStage = new SqlStageExecution(  
        queryId,  
        nextStageId,  
        locationFactory,  
        subStagePlan,  
        nodeScheduler,  
        remoteTaskFactory,  
        session,  
        splitBatchSize,  
        initialHashPartitions,  
        executor,  
        nodeTaskMap);  
    subStages.put(subStageFragmentId, subStage);  
}  
this.subStages = subStages.build();
```

6.2 查询调度过程

在介绍整个调度算法之前,也为了让读者更加清楚整个调度过程,本节深入 `SqlStageExecution` 内部组成,介绍几个重要组成部分:

(1) `NodeScheduler`: 将 task 分配给 node 的核心模块,其包含的功能有以下几个。

① `NodeManager`: 获取存活的节点列表,将其保存在 `NodeMap` 中,并且定时更新列表中的内容,缓存时长为 5 秒。

② `NodeMap`: 用于存储 Presto 节点信息,其包含了以下三部分内容。

- IP 和端口组成的节点列表。
- IP 组成的节点列表。
- Rack 组成的节点列表。

其中 Rack 仅是简单地将节点 `InetAddress` 封装成 `rackId`,并且一个节点对应一个 Rack,未接入类似于 Hadoop 中可配置机架感知脚本的功能。

③ `NodeSchedulerConfig`: 配置了调度的相关参数。

④ `NodeSelector`: 提供了各个 Stage 中 Task 分配节点的算法,详细的算法介绍请阅读 6.2.2 章节。

(2) `NodeTaskMap`: 保存了当前 Stage 分配的 Task 和节点的映射列表 `ConcurrentHashMap<Node, NodeTasks> nodeTasksMap`,其中 `NodeTasks` 维护了一个节点对应的 Task 列表,并且会对每个 Task 注册状态监听器,确保 Task 完成后去 Task 列表中移除。

(3) `RemoteTaskFactory`: 生成 `RemoteTask` 的工厂类。

(4) `StageStateMachine`: Stage 状态监听器。

6.2.1 NodeManager

Presto 调度 task 时要确保候选 node 都处于可工作状态, NodeManager 定义了统一的获取节点入口, 在 com.facebook.presto.server.CoordinatorModule 进行注册和绑定:

```
binder.bind(InternalNodeManager.class).to(DiscoveryNodeManager.class).in(Scopes.SINGLETON);
binder.bind(NodeManager.class).to(Key.get(InternalNodeManager.class)).in(Scopes.SINGLETON);
```

InternalNodeManager 除继承了 NodeManager 原有方法, 还新增了两个方法。所有方法均在 com.facebook.presto.metadata.DiscoveryNodeManager 实现, 具体的类关系以及定义方法如图 6-2 所示。

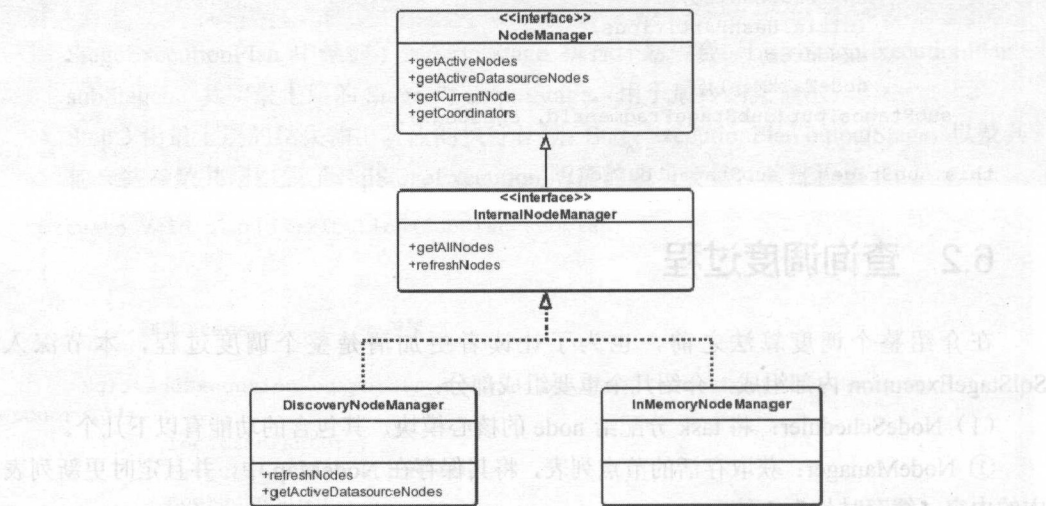


图 6-2

- NodeManager 定义的方法包括以下几个。
- Set<Node> getActiveNodes: 获取存活的节点列表。
 - Set<Node> getActiveDataSourceNodes: 根据 catalogName 获取存活列表。
 - Node getCurrentNode: 获取当前节点信息。
 - Set<Node> getCoordinators: 获取 Coordinator 列表。

InternalNodeManager 定义的方法包括以下几个。

- AllNodes getAllNodes: 获取所有的节点列表。
- refreshNodes: 刷新节点信息。

在调度过程中使用获取节点列表的核心方法为: refreshNodes。该方法会在获取节点的各种方法中进行调用, 调用频率为间隔 5 秒 (距离上一次更新时间)。Coordinator 间隔一定时间向各个 Work 节点发送 http 请求, 如果正常响应, 则表明该节点正常, 否则认为是异常节点。在 refreshNodes 方法中会把这些异常节点排除, 不参与 Task 的调度, 同时各个节点

的 Version 信息必须与 Coordinator 一致, 否则也认为是异常节点。再根据每个节点注册的 Catalog 信息, 在 Coordinator 上针对每个 Catalog 都会维护一个节点列表, 以供 getActiveDatasourceNodes 调用。

6.2.2 NodeSelector

NodeSelector 是整个查询调度的核心所在, 提供了各个 Stage 分配 Task 的具体算法。本节详细解析 NodeSelector 各个功能。

1. NodeMap

NodeSelector 维护了一个 NodeMap, NodeMap 本身根据节点信息维持了三份不同形式的列表。NodeMap 的节点信息通过调用 NodeManager.getActiveDatasourceNodes 获取, 采用了 Guava 的 com.google.common.base.Supplier 缓存技术:

```
public NodeSelector createNodeSelector(String dataSourceName)
{
    Supplier<NodeMap> nodeMap = Suppliers.memoizeWithExpiration(() -> {
        ImmutableSetMultimap.Builder<HostAddress, Node> byHostAndPort =
ImmutableSetMultimap.builder();
        ImmutableSetMultimap.Builder<InetAddress, Node> byHost =
ImmutableSetMultimap.builder();
        ImmutableSetMultimap.Builder<Rack, Node> byRack =
ImmutableSetMultimap.builder();
        Set<Node> nodes;
        if (dataSourceName != null) {
            // 根据数据源名称来获取对应的节点列表
            nodes = nodeManager.getActiveDatasourceNodes(dataSourceName);
        }
        else {
            nodes = nodeManager.getActiveNodes();
        }
        for (Node node : nodes) {
            try {
                byHostAndPort.put(node.getHostAndPort(), node);
                InetAddress host = InetAddress.getByName(node.getHttpUri().
getHost());
                byHost.put(host, node);
                byRack.put(Rack.of(host), node);
            }
            catch (UnknownHostException e) {
            }
        }
        return new NodeMap(byHostAndPort.build(), byHost.build(), byRack.build());
    }, 5, TimeUnit.SECONDS);
    return new NodeSelector(nodeMap);
}
```

目前 Rack (机架感知功能) 功能尚未实现, byHostAndPort 和 byHost 两个集合的目的是为了在 Source stage 分配 Split 时, 实现数据本地性而设置的, 详细的过程在下面会介绍。

2. Single 和 Fixed Stage 节点选择策略

Single 和 Fixed Stage 的节点选择比较简单，均为随机性选择，且调用方法为同一个：`NodeSelector.selectRandomNodes`。Single Stage 只会随机地从所有存活节点列表中选择一个节点，Fixed Stage 相比较于 Single Stage 略有不同，具体过程如下所示。

- 读取参数 `query.initial-hash-partitions` 的值，默认值为 8，`query.initial-hash-partitions` 的参数值决定了 Fixed Stage 参与的节点数，我们称之为 `FIXED_COUNT`。
- 在所有活跃的节点列表中随机选择 `query.initial-hash-partitions` 参数值指定的节点个数，我们称之为候选节点 `Candidates`，总的节点列表我们定义为 `allActiveNodes`。
- 如果 `allActiveNodes` 小于 `FIXED_COUNT`，此时候选节点 `candidates` 的个数并没有达到 `FIXED_COUNT` 所设置的值，`candidates` 值与 `allActiveNodes` 值一致。出现这种情况时，将根据 `node-scheduler.multiple-tasks-per-node-enabled` 参数值进行下一步判断，该参数的含义为“是否允许计算节点可分配多个 Task 任务”。但该参数也仅限于分配 Fixed stage 任务时才有效，在此将其定义为 `node_multiple_enable`。若 `node_multiple_enable` 为 `true`，那么将从候选节点 `candidates` 中再次选择，直到达到 `FIXED_COUNT` 的值为止；否则参与 Fixed Stage 的节点数只有 `candidates` 的个数，也就是说参与 Fixed Stage 的节点个数只能是 `allActiveNodes` 的数值，此时定义的 `FIXED_COUNT` 的值无效。

3. Source Stage 节点选择策略

每个查询请求都会含有一个或多个 Source Stage，一个 Table 会对应一个 Source Stage，Source Stage 的节点选择策略与 Single 和 Fixed Stage 完全不同，选择节点的个数是根据组成 Table 的 Split 个数来决定的。

从前面的章节我们知道，Presto Task 是由 Stage 来调度的，属于执行前调度。执行前调度表示整个查询在真正执行前都已分配完毕，不会在执行过程中更改 task 所在的计算节点。执行前调度有一个弊病，就是无法获知当前各个节点的资源使用情况，若调度不当，可能造成某些节点所分配到的 Split 非常多，导致数据倾斜，因此 Presto 为了避免这种情况，在分配 task 时采用的策略是尽量使每个节点处理的 Split 相对平均。

下面介绍调度前的准备工作以及相关参数。

- Source Stage 调度入口为 `SqlStageExecution.scheduleSourcePartitionedNodes`，从数据源所封装的 `SplitSource` 分批次（默认为 1000）地获取 source split，source split 定义了该数据源获取数据的目标以及范围。
- 每分配一个批次的 Source Split 时，将维护一个节点分配 Split 的计数器：`assignmentCount`，其含义为记录当前批次一个节点所分配到的 Split 个数。
- 计算出当前阶段每个节点的排队 Split 个数：`queuedSplitCount`，将用于候选节点的选举过程。
- 读取参数 `node-scheduler.location-aware-scheduling-enabled`，判断是否开启了数据本地性调度策略，若允许本地性调度策略，就根据本地性策略进行候选节点的选择，否

则将随机选择节点进行调度。

- 不论是否开启了本地性调度策略，首先都将选择出一批候选节点，候选节点的个数由参数 `node-scheduler.min-candidates` 决定，默认值为 10，针对每个 source split 将从候选节点中选择出满足条件的节点进行调度。

下面介绍候选节点的产生过程：

- Source split 会定义当前 Split 所描述的数据所在地址，例如，如果是 HIVE 数据源，那么数据所在地址就是 HDFS 的 block 所在地址。
- 如果 Split 定义了数据所在地址，首先能加入到候选节点列表的是能与 Split 数据所在 IP 和端口匹配的节点，其次可加入候选节点列表的是能与 Split 数据所在 IP 匹配的节点，这两个环节很有可能会匹配到同一个节点。
- 如果候选节点个数未达到 `node-scheduler.min-candidates` 的个数，将从 Rack 列表中继续挑选能匹配到 Split 数据所在 IP 的节点。
- 如果候选节点个数未达到 `node-scheduler.min-candidates` 所定义的值，并且允许远程读取，将采用随机性选择的策略来进行节点选择，以达到 `node-scheduler.min-candidates` 所设置的个数。
- 如果经过上述步骤，候选节点列表为空，那么判断当前 Split 数据所在 IP 是否包含了 Coordinator，如果是，那么就选择 Coordinator 作为候选节点；否则就直接返回空列表；如果最终的候选节点为空列表，则系统抛错，查询失败。

获取到候选节点列表后，接下来将根据以下原则选择节点。尽量将待分配的 Split 分配给运行最少 Split 的节点进行处理，详细过程如下。

- 计算出每个候选节点当前历史已分配的 Split 个数：`splitCountByNode`。
- 从候选节点中选择出“历史已分配 Split 个数 (`splitCountByNode`)”与“当前批次已分配的 Split 个数 (`assignmentCount`) 之和”较小的节点，也就是 $\min(\text{splitCountByNode}, \text{assignmentCount})$ 。两者之和定义为 `totalRunningSplitCount`，但 `totalRunningSplitCount` 要小于每个节点允许运行的最大 Split 个数，每个节点允许运行最大 Split 个数由参数 `node-scheduler.max-splits-per-node` 来配置。
- 如果上述步骤未选择出满足条件的节点，则选择“排队的 Split 个数 (`queuedSplitCount`)”与“当前批次已分配的 Split 个数 (`assignmentCount`) 之和”较小的节点，即为 $\min(\text{queuedSplitCount}, \text{assignmentCount})$ ，两者之和定义为 `totalPendingSplitCount`，但也需要满足另外一个条件：`totalPendingSplitCount` 要小于每个节点允许排队的 Split 最大个数，每个节点允许排队的 Split 最大个数由参数 `node-scheduler.max-pending-splits-per-node-per-task` 来配置。
- 选择节点后，更新计数器 `assignmentCount`，继续下一个 Split 的分配，直至当前批次所有的 Split 全部分配完毕。

Source stage 的调度过程其实就是为 Split 分配节点的过程，其遵循的原则是：尽量使每个节点所分配到的 Split 个数相对平均，但这种策略可能会导致以下两个情况：

- 虽然 Split 定义了数据所在地址，但是有可能该 Split 无法分配到该地址对应的节点，

就算 Split 数据所在节点包含在候选节点列表中，但是也有可能不会被选择到，因为该节点运行的 Split 可能已经很多了。

- 尽量使每个节点所分配到的 Split 个数相对平均的原则是基于整个 Presto 集群层面的考虑，当提交的查询比较多时，具体到单个查询时候规则可能将会被打破。例如某个时刻一起向 Presto 集群提交了两个查询请求：query1 和 query2，简单分析 Presto 集群中两个节点（node1 和 node2）的分配情况，此时 source stage 的调度策略只会考虑 node 层面 Split 个数是否相对均匀，可能出现的局面为 query1 在 node1 上分配了 5 个 Splits，在 node2 上分配了 15 个 Splits；而 query2 在 node1 上分配了 15 个 Splits，在 node2 上分配了 5 个 Splits。虽然在整个 Presto 层面上看两个节点分配的 Split 个数都是 20 个 Splits，此时两个节点分配的 Split 也是相对平均的，但具体到 query1 和 query2 两个查询请求的执行来说，分配结果均不理想。

6.3 小结

Presto 的查询调度本质上就是 Split 分配到各个节点的过程，每个阶段依据本身所承担的职责，调度方式有所区别，从整体上来说，Split 分配节点的方式基本为随机选择的策略，在此基础上尽量保证每个节点处理的 Split 相对平均。

从 Presto 性能方面考虑，每个阶段的每个计算节点处理的 Split 个数也在很大程度上决定了查询的性能，因此在调整查询调度的各个参数配置时，需要根据集群规模以及处理的数据量来衡量各个参数的值。

第 7 章

查询执行

向 Presto 集群提交一个查询，其整个执行过程会经历以下 4 个阶段。

- (1) 提交查询：客户端向 Coordinator 提供的 RESTful 服务提交 SQL 语句。
- (2) 生成查询执行计划：Coordinator 根据传递的 SQL 语句生成相应的查询执行计划。
- (3) 查询调度：Coordinator 根据生成的查询执行计划，依次进行 Stage 和 Task 的调度。
- (4) 查询执行：最终 Coordinator 会调度最空闲的 Worker 执行相应的 Task 进行实际的计算任务。

综上所述，在 Presto 中执行查询的整体流程如图 7-1 所示。

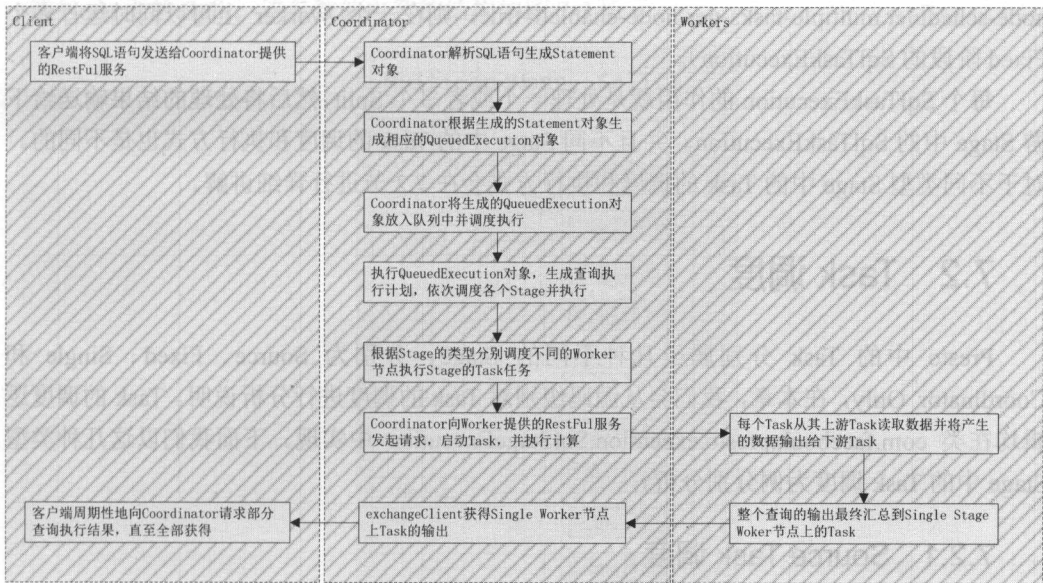


图 7-1

7.1 查询执行逻辑

通过前面的章节可知，Presto 中的所有查询会依次转化为 `SqlQueryExecution`、`SqlStageExecution` 和 `SqlTaskExecution`，因此所有的查询最终都会转化成为一个个单独的 Task 在每个 Worker 节点上执行，如图 7-2 所示。

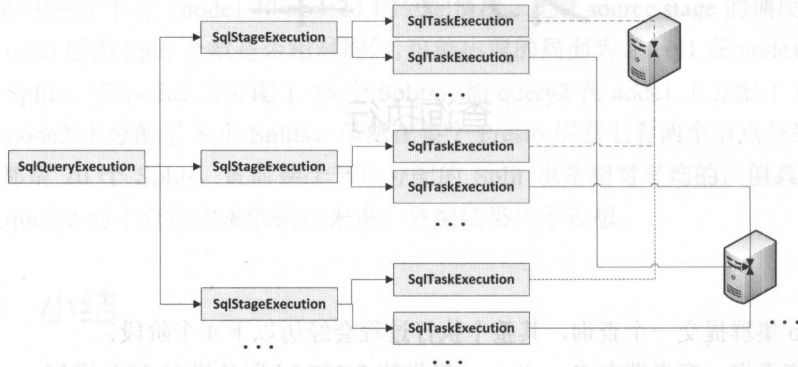


图 7-2

从图 7-2 中可以看出，一个 `SqlQueryExecution` (Query) 可以被分解成多个 `SqlStageExecution`，而一个 `SqlStageExecution` (Stage) 可以被分解成多个 `SqlTaskExecution` (Task)，每个 `SqlTaskExecution` 运行在一个 Worker 上，并且在一个 Worker 上运行的属于同一个 `SqlStageExecution` 的 `SqlTaskExecution` 个数不能多于一个（当然在特殊情况下，若设置 `node-scheduler.multiple-tasks-per-node-enabled=true`，则在一个 Worker 上可以同时运行多个 Fixed 阶段的 `SqlTaskExecution`）。

每个 `SqlTaskExecution` 的作用就是处理一个或者多个 Split，然后将处理的结果输送给下游 Stage 中的 `SqlTaskExecution`。并且不同 Stage 中的 Task 的调度和执行方式也是不同的。对于不同类型 Stage 中的 Task 的调度和执行方式会在 7.2 节进行详细讲解。

7.2 Task 调度

Presto 中的 Task 分别属于几种不同的 Stage，分别为 Source、Fixed、Single 和 `Coordinator_Only`，在本章，我们会从源码级别对 Task 的调度进行分析说明。Task 的调度逻辑均在类 `com.facebook.presto.execution.SqlStageExecution` 中实现。下面我们以这几种类型 Stage 中的 Task 调度为例分别分析。

7.2.1 Source Task 调度

我们从前面的章节已经了解到，Presto 可以读取多种数据源中的数据，然后在内存中进行计算。Presto 就是通过 Source Stage 从各种各样的数据源中读取数据的，属于 Source Stage 的 Task 我们称之为 Source Task，Source Task 的执行逻辑如下。

- (1) 根据 Split 的本地性，生成 Node 与 Split 的对应关系。
- (2) 根据 Node 与 Split 的对应关系，在指定的 Node 上启动 Source Task 处理位于该 Node 上的 Splits。

上述执行逻辑在代码中的实现过程如下：

```
private void scheduleSourcePartitionedNodes() throws InterruptedException {
    .....
    while (!pendingSplits.isEmpty() && !getState().isDone()) {
        //获得 Node 与 Split 的对应关系，其内容形式为：Node->Set<Split>，即：每个 Node 上存在
        哪些 Split
        Multimap<Node, Split> splitAssignment =
        nodeSelector.computeAssignments(pendingSplits, tasks.values());
        .....
        //该方法用于在指定的 Node 上启动 Task 或者更新指定 Node 上的已经存在的 Task 需要处理的
        Splits
        assignSplits(nextTaskId, splitAssignment);
        .....
    }
}
```

上面的代码主要用方法 assignSplits(nextTaskId, splitAssignment)来完成 Source Task 的启动和执行，该方法的传入实际参数的意义介绍如下。

nextTaskId: 实际上是一个整数，用于根据该参数生成新 Task 的 Id。

splitAssignment: 该参数的类型为 Multimap<Node, Split>，用于记录 Node 与 Splits 的一对多的映射关系，每一个 Node 对应一个或者多个 Split，其对应的 Split 实际上就是位于该 Node 上的 Split。该参数的内容的结构如图 7-3 所示。

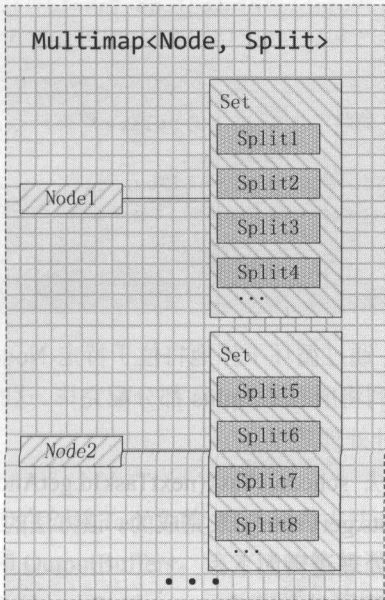


图 7-3

该方法对传入的参数 `splitAssignment` 进行遍历，并对每个 `Entry` 都执行如下操作，直到将 `splitAssignment` 中的所有 `Entry` 都遍历完毕。

(1) 根据 `Node` 获得该 `Node` 上的 `Task`。

(2) 若 `Task` 为空，则根据 `nextTaskId`、`Node` 以及该 `Node` 上的 `Splits` 创建一个新的 `Task`；否则将该 `Node` 上的 `Splits` 提交给运行在该 `Node` 上的 `Task` 进行处理（这种提交的操作是通过 `RESTful` 服务完成的，在后面我们会详细介绍）。

上述执行逻辑的代码如下：

```
private void assignSplits(AtomicInteger nextTaskId, Multimap<Node, Split>
splitAssignment) {
    //遍历 splitAssignment
    for (Entry<Node, Collection<Split>> taskSplits :
splitAssignment.asMap().entrySet()) {
        Node node = taskSplits.getKey();
        //根据当前 Stage 的 Node 与 TaskId 的映射关系获得特定 Node 对应的唯一的 TaskId，若存
        在则直接返回该 TaskId，否则返回 null。从这里可以看出 Stage 中的 Node 与 Task 是一一对
        应的
        TaskId taskId = Iterables.getOnlyElement(localNodeTaskMap.get(node), null);
        RemoteTask task = taskId != null ? tasks.get(taskId) : null;
        if (task == null) {
            //若获得的 Task 为 null，则说明在 Node 上没有启动 Task，则在 Node 上启动一个 Task 运行，该 Task
            用于处理 taskSplits 中的所有 Splits
            RemoteTask remoteTask = scheduleTask(nextTaskId.getAndIncrement(), node,
            fragment.getPartitionedSource(), taskSplits.getValue());
            //通知上游 Stage 为该 Task 创建或者更新 OutputBuffers，上游 Stage 通过 OutputBuffers 向该
            Task 传递数据，但是由于该 Task 属于 Source Stage，因此没有上游 Stage，因此该方法不会执行任何操作，
            但是若 Task 属于 Single 或者 Fixed Stage，则其上游 Stage 可以创建或者更新 OutputBuffers
            addStageNode(remoteTask.getTaskInfo().getTaskId());
            stateMachine.recordScheduleTaskTime(scheduleSplitStart);
        }
        else {
            //若获得的 Task 不为 null，则将当前 Node 上的 Splits 更新到运行在该 Node 上的 Task 中需要处理
            的 Split 列表中
            task.addSplits(fragment.getPartitionedSource(), taskSplits.getValue());
            stateMachine.recordAddSplit(scheduleSplitStart);
        }
    }
}
```

从上面的代码可以看出，通过循环不停地将位于特定 `Node` 上的 `Split` 均交给运行在该 `Node` 上的 `Task` 进行处理，直到将所有的 `Splits` 都分配完毕。这时，运行在各个节点上的 `Task` 均会处理本地 `Split`，从而提升处理效率。

新 `Task` 的创建是由方法 `scheduleTask(nextTaskId.getAndIncrement(), node, fragment.getPartitionedSource(), taskSplits.getValue())` 来完成的。该方法的主要作用就是在指定的 `Node` 上启动一个 `Task` 并执行，其处理逻辑如下。

(1) 更新当前 `Stage` 的所有 `exchange` 和 `outputBuffers`。从前面的章节我们已经知道：对每个 `Stage` 来说，`exchange` 用于从其上游 `Stage` 中获得数据，而 `outputBuffer` 则用于将当前

Stage 的数据输出给下游 Stage。

(2) 将传入的 Split 与上游 Stage 中的每个 Task 的输出组合起来作为整体的 Split 集合: initialSplits。

(3) 根据 initialSplits、Node 和 taskId 等相关信息, 在 Node 上启动一个 Task 进行 initialSplits 的处理。

该方法的代码如下所示:

```
private RemoteTask scheduleTask(int id, Node node, PlanNodeId sourceId, Iterable<?
extends Split> sourceSplits)
{
    // 在创建一个新的 Task 之前, 首先要更新所有的 exchange 和 outputBuffers
    addNewExchangesAndBuffers();
    //根据转入的整数值生成一个 taskId
    TaskId taskId = new TaskId(stateMachine.getStageId(), String.valueOf(id));
    ImmutableMultimap.Builder<PlanNodeId, Split> initialSplits =
    ImmutableMultimap.builder();
    //首先收集所有的 SourceSplit, 在类型为 Source 的 Stage 中, 该方法的传入参数:
    sourceSplits 是有值的; 而在类型为 Fixed 或者 Single 的 Stage 中, 该方法的传入参数: sourceSplits
    其实是没有值的
    for (Split sourceSplit : sourceSplits) {
        initialSplits.put(sourceId, sourceSplit);
    }
    // PlanNodeId 标示一个 Stage, URI 标示一个 TaskLocation。遍历每个上游 Stage 中的每
    个 Task, 针对每个 Task 组装成为一个 Split, 并添加进 initialSplits 中
    for (Entry<PlanNodeId, URI> entry : exchangeLocations.get().entries()) {
        initialSplits.put(entry.getKey(), createRemoteSplitFor(taskId,
        entry.getValue()));
    }
    //在 Node 上创建一个 Task
    RemoteTask task =
    remoteTaskFactory.createRemoteTask(stateMachine.getSession(),
        taskId,
        node,
        fragment,
        initialSplits.build(),
        getCurrentOutputBuffers());
    .....
    //在 start() 方法中开始真正执行 Task 并处理 initialSplits 集合中的 Split
    task.start();
    .....
    //更新当前 Stage 状态
    doUpdateState();
    return task;
}
```

从上面的代码中可以看到, 在创建 Task 之前, 首先通过方法 addNewExchangesAndBuffers() 更新当前 Stage 的 Exchange 和 OutputBuffers。该方法的执行逻辑如下。

(1) 更新并获取最新的 ExchangeLocations, 并将其与现有的 ExchangeLocations 合并到一起。

(2) 更新并获取当前 Stage 最新的 OutputBuffers, 因为当前 Stage 的 OutputBuffers 有可能会由其下游 Stage 调用的 `parentTasksAdded(List<TaskId> parentTasks, boolean noMoreParentNodes)` 方法进行更新, 因此每次都需要更新并获取最新的 outputBuffers。

(3) 遍历当前 Stage 的所有 Task, 将对每个 Task 执行下述一系列操作。

① 遍历最新的 ExchangeLocations, 根据其上游 Stage 中的每个 Task Location 生成一个 Split, 并更新到 Task 处理的 Split 列表中。

② 将最新的 outputBuffers 更新到 Task 中。

上述执行逻辑的代码如下所示:

```
private boolean addNewExchangesAndBuffers()
{
    .....
    //获得最新的 newExchangeLocations, 并更新到当前 Stage 现有的 exchangeLocations 中
    Multimap<PlanNodeId, URI> newExchangeLocations=getNewExchangeLocations();
    exchangeLocations.set(ImmutableMultimap.<PlanNodeId, URI>builder()
        .putAll(exchangeLocations.get())
        .putAll(newExchangeLocations)
        .build());
    //更新并获取当前 Stage 最新的 OutputBuffers
    OutputBuffers outputBuffers = updateToNextOutputBuffers();
    .....
    //遍历当前 Stage 的各个 Task, 并以最新的 ExchangeLocations 和 OutputBuffers 对其进行更新
    try (SetThreadName ignored = new SetThreadName("SqlStageExecution-%s",
        stateMachine.getStageId())) {
        for (RemoteTask task : tasks.values()) {
            for (Entry<PlanNodeId, URI> entry : newExchangeLocations.entries())
            {
                Split remoteSplit = createRemoteSplitFor(task.getTaskInfo().
                    getTaskId(), entry.getValue());
                task.addSplits(entry.getKey(),
                    ImmutableList.of(remoteSplit));
            }
            task.setOutputBuffers(outputBuffers);
            completeSources.forEach(task::noMoreSplits);
        }
    }
    .....
}
```

从上面的代码可以看出, 分别通过方法 `getNewExchangeLocations()` 和 `updateToNextOutputBuffers()` 来获得最新的 `exchangeLocations` 和 `outputBuffers`。下面分别对这两个方法进行分析和说明。

从前面的章节可以知道, Presto 中的每个 Stage 都需要从上游 Stage 中获得输出作为自己的输入, 因此每个 Stage 都需要记住自己的上游 Stage 是哪些, 并且从某个上游 Stage 获得其输出的时候需要知道从上游 Stage 的哪个 Task 去获取。在每个 Stage 中都是通过全局变量 `exchangeLocations` 来记录这些关系的, `exchangeLocations` 的类型为 `AtomicReference<Multimap<PlanNodeId, URI>>`。其中 `PlanNodeId` 类型的 key 用于标示当前

Stage 的上游 Stage, URI 类型的 value 用于标示上游 Stage 中的某个 Task, 一个 PlanNodeId 对应多个 URI, Multimap 的数据结构与图 7-3 所示的数据结构非常类似。

方法 `getNewExchangeLocations()` 用于获取最新的 `exchangeLocations`, 其执行逻辑如下。

遍历当前 Stage 的上游 Stage 中的各个 Task, 并检查该 Task 的 TaskLocation 对应的 URI 是否包含在当前 Stage 的 `exchangeLocations` 中, 若包含在其中, 则跳过当前操作, 继续遍历下一个 Task; 否则, 就将 Task 的 TaskLocation 对应的 URI 加入到当前 Stage 的 `exchangeLocation` 中。该方法执行逻辑的代码如下:

```
private Multimap<PlanNodeId, URI> getNewExchangeLocations() {
    Multimap<PlanNodeId, URI> exchangeLocations = this.exchangeLocations.get();
    ImmutableMultimap.Builder<PlanNodeId, URI> newExchangeLocations =
    ImmutableMultimap.builder();
    //遍历上游 stage 的所有 Task, 并将每个 Task 的 TaskLocation 对应的 URI 加入到当前 Stage
    的 exchangeLocations 中
    for (RemoteSourceNode remoteSourceNode : fragment.getRemoteSourceNodes()) {
        for (PlanFragmentId planFragmentId :
        remoteSourceNode.getSourceFragmentIds()) {
            SqlStageExecution subStage = subStages.get(planFragmentId);
            for (URI taskLocation : subStage.getTaskLocations()) {
                if (!exchangeLocations.containsEntry(remoteSourceNode.getId(),
                taskLocation)) {
                    newExchangeLocations.putAll(remoteSourceNode.getId(),
                    taskLocation);
                }
            }
        }
    }
    return newExchangeLocations.build();
}
```

方法 `updateToNextOutputBuffers()` 用于获取最新的 `OutputBuffers`, 之所以要获取最新的 `OutputBuffers`, 是因为当前 Stage 的 `OutputBuffers` 有可能会因为其下游 Stage 调用了方法 `parentTasksAdded(List<TaskId> parentTasks, boolean noMoreParentNodes)` 而更改当前 Stage 的 `OutputBuffers` 对象。在对其执行逻辑进行说明之前, 首先需要对 `SqlStageExecution` 中的两个全局变量进行说明。

- `nextOutputBuffers`: 指向通过方法 `parentTasksAdded(List<TaskId> parentTasks, boolean noMoreParentNodes)` 修改之后的最新的 `OutputBuffers`。
- `currentOutputBuffers`: 指向 `parentTasksAdded(List<TaskId> parentTasks, boolean noMoreParentNodes)` 方法修改之前的 `OutputBuffers`。

该方法的执行逻辑为: 若 `nextOutputBuffers` 为空, 则直接返回 `currentOutputBuffers`; 否则将 `nextOutputBuffers` 赋给对象 `currentOutputBuffers`, 然后将 `nextOutputBuffers` 置为 `null`, 最后返回 `currentOutputBuffers`。该方法的执行代码如下:

```
private synchronized OutputBuffers updateToNextOutputBuffers() {
    if (nextOutputBuffers == null) {
        return currentOutputBuffers;
    }
}
```



```

    }
    currentOutputBuffers = nextOutputBuffers;
    nextOutputBuffers = null;
    this.notifyAll();
    return currentOutputBuffers;
}

```

7.2.2 Fixed Task 调度

属于 Fixed Stage 的 Task 我们称之为 Fixed Task。从前面的章节可以得知：在 Fixed Stage 中会将 Join 操作两边的数据集分别分解成 initial-hash-partitions 个数的数据子集，然后将 Hash 值相等的行分布到相同的 Worker 上进行 Join 计算。这样就相当于以 initial-hash-partitions 倍的并行度进行计算，从而极大地提高了 Fixed Stage 的计算性能。运行在每个 Worker 上的 Fixed Task 的任务就是将 Hash 过来的数据进行 Join 操作，然后将结果输出给下游 Stage。

从前面的章节我们可以知道，通过随机的方式选择了需要运行 Fixed Task 的 Node，那么后续的工作就是在对应的 Node 上启动 Task，并通知上游 Stage，创建或者更新上游 Stage 的 OutputBuffers，上游 Stage 产生的数据通过 OutputBuffers 输出给当前 Stage 中的 Task。具体的代码逻辑如下：

```

private void scheduleFixedNodeCount(int nodeCount)
{
    // 随机选择 nodeCount 个 Node
    List<Node> nodes = nodeSelector.selectRandomNodes(nodeCount);
    checkCondition(!nodes.isEmpty(), NO_NODES_AVAILABLE, "No worker nodes
available");
    ImmutableList.Builder<TaskId> tasks = ImmutableList.builder();
    //遍历所有随机选择到的 Node，并在每个 Node 上启动一个 Task
    for (int taskId = 0; taskId < nodes.size(); taskId++) {
        Node node = nodes.get(taskId);
        //在指定的 Node 上启动 ID 为 taskId（在这里其实就是一个整数，后面会根据 Stage 的
ID 组装成我们在 Web 页面上看到的 TaskId）的 Task
        RemoteTask task = scheduleTask(taskId, node);
        tasks.add(task.getTaskInfo().getTaskId());
    }

    //通知上游 Stage，创建或者更新 OutputBuffers
    for (SqlStageExecution subStage : subStages.values()) {
        subStage.parentTasksAdded(tasks.build(), true);
    }
}

```

从上面的代码可以看出，对 Task 的创建和启动，均由方法 scheduleTask(taskId, node) 来完成，其实该方法只是对方法 private RemoteTask scheduleTask(int id, Node node, PlanNodeId sourceId, Iterable<? extends Split> sourceSplits)的一个封装，最终会调用方法 private RemoteTask scheduleTask(int id, Node node, PlanNodeId sourceId, Iterable<? extends Split> sourceSplits)，由于在 7.2.1 节中已经对该方法进行了说明，因此这里就不再赘述了。除此之外，需要通过方法 subStage.parentTasksAdded(tasks.build(), true)通知上游 Stage，针对

当前 Fixed Stage 的所有 Task 创建或者更新 OutputBuffers，用于将上游 Stage 产生的结果输出给相应的 Fixed Task。该方法的执行逻辑为：判断当前 Stage（相对于 Fixed Stage 来说，就是上游 Stage）的输出是否需要进行 Hash，若不需要进行 Hash，则针对于每个 taskId 创建一个 UnpartitionedPagePartitionFunction 对象，并将该对象放入到 newBuffers 中；否则针对每个 taskId 创建一个 HashPagePartitionFunction，并将该对象放入到 newBuffers 中。最终会针对所有的 parent task（即 Fixed task）生成更新后的 nextOutputBuffers，此 nextOutputBuffers 最终会用于将 subStage 中的 Task 生成的数据输出给 Fixed Stage 中的 Task。parentTasksAdded 方法的代码如下：

```
private synchronized void parentTasksAdded(List<TaskId> parentTasks, boolean
noMoreParentNodes) {
    // 若全局变量 nextOutputBuffers 不为空，则以 nextOutputBuffers 作为初始
    outputbuffers；否则以 currentOutputBuffers 作为 outputbuffers
    OutputBuffers startingOutputBuffers = nextOutputBuffers != null ?
nextOutputBuffers : currentOutputBuffers;
    // 添加新的 outputbuffers
    OutputBuffers newOutputBuffers;
    //若当前 Stage 的输出不需要进行 Hash 分布，则针对每个 taskId 创建一个
    UnpartitionedPagePartitionFunction 对象
    if (fragment.getOutputPartitioning() == OutputPartitioning.NONE) {
        ImmutableMap.Builder<TaskId, PagePartitionFunction> newBuffers =
ImmutableMap.builder();
        for (TaskId taskId : parentTasks) {
            newBuffers.put(taskId, new UnpartitionedPagePartitionFunction());
        }
        newOutputBuffers =
startingOutputBuffers.withBuffers(newBuffers.build());
        // no more flag
        if (noMoreParentNodes) {
            newOutputBuffers = newOutputBuffers.withNoMoreBufferIds();
        }
    }
    //若当前 Stage 的输出需要进行 Hash 分布，则针对每个 taskId 创建一个
    HashPagePartitionFunction 对象
    else if (fragment.getOutputPartitioning() == OutputPartitioning.HASH) {
        ImmutableMap.Builder<TaskId, PagePartitionFunction> buffers =
ImmutableMap.builder();
        for (int nodeIndex = 0; nodeIndex < parentTasks.size(); nodeIndex++)
        {
            TaskId taskId = parentTasks.get(nodeIndex);
            buffers.put(taskId, new HashPagePartitionFunction(nodeIndex,
parentTasks.size(), getPartitioningChannels(fragment), getHashChannel(fragment),
fragment.getTypes()));
        }
        newOutputBuffers = startingOutputBuffers
            .withBuffers(buffers.build())
            .withNoMoreBufferIds();
    }
    else {
```

```
        throw new UnsupportedOperationException("Unsupported output
partitioning " + fragment.getOutputPartitioning());
    }
    .....
}
```

7.2.3 Single Task 调度

属于 Single Stage 的 Task 我们称之为 Single Task。从前面的章节可以得知，在 Single Stage 中会将其上游 Stage 的所有输出都汇集到一个 Worker 节点上进行汇总计算。运行在 Worker 上的 Single Task 的任务就是将其从上游 Stage 中所有 Task 产生的数据进行汇总计算，然后将结果输出给下游 Stage 或者输出给 Coordinator，最后再由 Coordinator 输出给客户端。Single Task 典型的使用场景就是全局排序，在进行全局排序计算的时候，是在 Single Stage 中完成的。在进行计算之前会将其上游 Stage 的输出全部都汇集到 Single Node 上，然后在 Single Node 上运行的 Single Task 会将汇聚到的数据进行全局排序。

Single Task 的调度逻辑与 Fixed Task 的调度逻辑完全一致，只不过 Fixed Stage 有多个 Task，而 Single Stage 有一个 Task。其执行方法也是 scheduleFixedNodeCount(int nodeCount)，只不过传入此方法的参数为 1。关于该方法的执行逻辑在 7.2.2 节我们已经进行了详细说明，因此就不在此赘述了。

7.2.4 Coordinator_Only Task 调度

当执行 DDL 或者 DML 语句的时候，只需要在 Presto 的 Coordinator 对元数据进行添加或者更改。

另外，当执行 Create Table As Select 语句的时候，将会产生两个 Stage，分别是 Source Stage 和 Coordinator_Only Stage，其中 Source Stage 负责从表中查询语句并将数据写入表对应的路径下；Coordinator_Only Stage 负责根据查询语句创建元数据。其查询执行计划如图 7-4 所示。

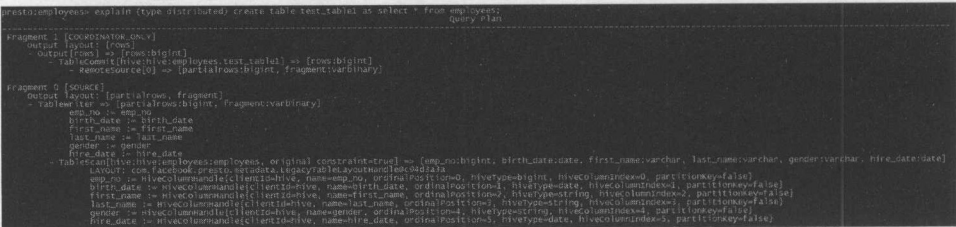


图 7-4

从图 7-4 中可以看到，执行 Create Table As Select 语句时确实如上所述分为了 Source 和 Coordinator_Only 两个 Stage，而在 Cli 或者页面上显示的插入的记录条数是由 Source Stage 的多个 Task 上的 partialRows 累加起来得到的，其实际代码在类 TableCommitOperator 中，如下所示：

```
public void addInput(Page page) {
    Block rowCountBlock = page.getBlock(0);
```

```

.....
for (int position = 0; position < page.getPositionCount(); position++) {
    //将各个 partialRows 累加起来得出总数
    if (!rowCountBlock.isNull(position)) {
        rowCount += BIGINT.getLong(rowCountBlock, position);
    }
    .....
}
}
}

```

无论是直接执行 DDL 或 DML, 还是执行 Create Table As Select... 语句, Coordinator_Only Stage 的 Task 都是运行在 Coordinator 上的, 而且 Coordinator_Only Stage 中只有一个 Task, 该 Task 的唯一作用就是创建或者更新元数据并总结查询的总条数。Coordinator_Only 类型的 Task 的调度代码如下:

```

private void scheduleOnCurrentNode()
{
    //获取当前节点。由于 Stage 和 Task 的调度代码都运行在 Coordinator 上, 因此当前节点就是 Coordinator 节点
    Node node = nodeSelector.selectCurrentNode();
    //直接在 Coordinator 上启动一个 Task
    RemoteTask task = scheduleTask(0, node);
    .....
}

```

从上面的代码可以看出, Coordinator_Only Stage 在选取节点的时候是选取了 Coordinator 节点, 而 Single Stage 是随机选取节点, 而且都是调用 scheduleTask(int id, Node node) 方法创建 Task, 因此就不再赘述其他代码了。

7.3 Task 执行

在前面的章节中详细说明了各种类型的 Task 的调度过程, 我们发现, 无论是哪种类型的 Task, 最终都需要调用方法 scheduleTask(int id, Node node, PlanNodeId sourceId, Iterable<? extends Split> sourceSplits) 创建 Task, 或者调用方法 HttpRemoteTask.addSplits(PlanNodeId sourceId, Iterable<Split> splits) 更新 Task。因此本节主要从这两个方法入手, 对 Task 的执行进行详细讲解。

7.3.1 创建 Task

从上面的代码可以看出, Coordinator_Only、Single 和 Fixed 类型的 Task, 在 Task 调度节点都是直接创建 Task, 而 Source 类型的 Task, 由于其分批调度 Splits, 因此有可能会多次调用 scheduleTask 方法。所以, 在调用 scheduleTask 方法的时候会判断对应的 Node 上是否已经创建了 Task: 若已经创建, 就更新 Task; 否则就创建 Task。其实在创建 Task 的时候, 主要是调用 HttpRemoteTask 类的构造方法创建一个 HttpRemoteTask 对象, 并调用

HttpRemoteTask 对象的 Start()方法,而 Start()方法最终又会调用封装在 HttpRemoteTask 内部的 httpClient,向特定的 Worker Node 上的 TaskResource 服务发起 RESTful 请求,从而在特定的 Worker Node 上启动一个对应的 SqlTaskExecution 进行数据处理和计算。因此创建 Task 分为两部分:

- (1) RESTful Client 端创建 HttpRemoteTask 对象。
- (2) RESTful Resource 端创建 SqlTaskExecution 对象。

下面依次对这两部分进行讲解。

1. Client 端

在 Client 端创建 Task 的过程分为两步:创建 HttpRemoteTask 对象和调用该对象的 Start()方法,相关代码如下:

```
// 创建 HttpRemoteTask 对象, RemoteTask 只是一个接口, 其实现类为: HttpRemoteTask
RemoteTask task = remoteTaskFactory.createRemoteTask(stateMachine.getSession(),
    taskId,
    node,
    fragment,
    initialSplits.build(),
    getCurrentOutputBuffers());
//为 task 注册 listener, 每当状态发生改变都调用 doUpdateState 方法进行状态更新
task.addStateChangeListener(taskInfo -> doUpdateState());
//调用 HttpRemoteTask 对象的 start()方法, 向 Worker Node 节点上的 TaskResource 服务发起
RESTful 请求, 从而在 Node 上创建一个 SqlTaskExecution 进行实际的计算和处理
task.start();
```

从上面的代码可以看出,创建 Task 的 Client 端操作主要包括 HttpRemoteTask 对象的创建和 HttpRemoteTask 对象的 Start()方法的调用,下面针对这两步操作进行详细讲解。

(1) 创建 HttpRemoteTask 对象

创建 HttpRemoteTask 对象的调用堆栈如图 7-5 所示,可以看到 4 种 Task(Source、Fixed、Single 和 Coordinator_Only)的调度都会调用 HttpRemoteTask 对象的构造方法。在 HttpRemoteTask 对象中保存了 Task 对应的 taskId、nodeId、outputBuffers、需要处理的 Splits 和用于发起 RestFul 请求的 httpClient 等。这些信息在调用 HttpRemoteTask 对象的.Start()方法时会被用来组建和发送 RESTful 请求。

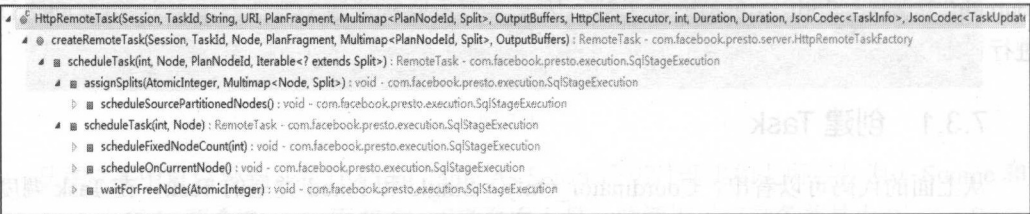


图 7-5

最终会调用到 HttpRemoteTask 对象的构造方法,其构造方法如下:

```
public HttpRemoteTask(Session session,
```



```

TaskId taskId,
String nodeId,
URI location,
PlanFragment planFragment,
Multimap<PlanNodeId, Split> initialSplits,
OutputBuffers outputBuffers,
HttpClient httpClient,
Executor executor,
int maxConsecutiveErrorCount,
Duration minErrorDuration,
Duration refreshMaxWait,
JsonCodec<TaskInfo> taskInfoCodec,
JsonCodec<TaskUpdateRequest> taskUpdateRequestCodec)
{
    .....
    try (SetThreadName ignored = new SetThreadName("HttpRemoteTask-%s",
taskId)) {
        //创建的 Task 对应的 taskId
        this.taskId = taskId;
        //对应的 session 参数
        this.session = session;
        //需要启动 SqlTaskExecution 的 Node 的 NodeId
        this.nodeId = nodeId;
        //planFragment 是对当前 Stage 的抽象描述, 其中记录了当前 Stage 的大部分信息
        this.planFragment = planFragment;
        //当前 Stage 的 outputBuffers, outputbuffers 用于将当前 Stage 的结果输出给下
        //游 Stage
        this.outputBuffers.set(outputBuffers);
        //httpClient 将会在 start() 方法中向 TaskResource 发送 RestFul 请求
        .....
        //遍历所有的 initialSplits, 并将所有的 Split 对象封装成 ScheduledSplit 对象,
        //然后放入到 pendingSplits 中, 供 start() 方法使用
        for (Entry<PlanNodeId, Split> entry : checkNotNull(initialSplits,
"initialSplits is null").entries()) {
            ScheduledSplit scheduledSplit = new
ScheduledSplit(nextSplitId.getAndIncrement(), entry.getValue());
            pendingSplits.put(entry.getKey(), scheduledSplit);
        }
        //记录当前 Stage 处理上游 Stage 的 Split 的个数
        if (initialSplits.containsKey(planFragment.getPartitionedSource())) {
            pendingSourceSplitCount =
initialSplits.get(planFragment.getPartitionedSource()).size();
        }
        .....
        // ContinuousTaskInfoFetcher 用于不停地从 TaskResource 处获得当前 Task 的信息
        continuousTaskInfoFetcher = new
ContinuousTaskInfoFetcher(refreshMaxWait);
    }
}

```

(2) 调用 Start()方法

当创建完毕 HttpRemoteTask 对象之后, 就需要调用该对象的 start()方法向 TaskResource 发送 RESTful 请求, 在相应的节点上创建一个 SqlTaskExecution 对象, 从而进行真正的 Split

计算和处理。其相关的代码如下：

```
public void start() {
    try (SetThreadName ignored = new SetThreadName("HttpRemoteTask-%s", taskId))
    {
        //想要启动 Task，我们只需要触发 update 即可
        scheduleUpdate();
        // 开始持续地从 TaskResource 处获得 Task 的最新信息
        continuousTaskInfoFetcher.start();
    }
}
```

从上面的代码可以看出，启动 Task 的主要代码都集中在方法 `HttpRemoteTask` 对象的 `scheduleUpdate()` 中，该方法的核心代码如下所示：

```
private synchronized void scheduleUpdate() {
    //若 Task 还未开始执行或者 Task 已经结束了，则不需要做任何操作，直接从方法返回
    if (!needsUpdate.get() || taskInfo.get().getState().isDone()) {
        return;
    }

    .....
    //将当前 Task 需要处理的所有 Split 都组装成 TaskSource 列表
    List<TaskSource> sources = getSources();
    //使用 TaskSource 列表、outputbuffers 创建 request 请求
    TaskUpdateRequest updateRequest = new TaskUpdateRequest(session,
        planFragment,
        sources,
        outputBuffers.get());
    //生成一个地址为：http://WorkerNodeIP:Port/v1/task/taskId?summarize 的 Post 请求
    Request request = preparePost()
        .setUri(uriBuilderFrom(taskInfo.get().getSelf()).addParameter("summarize").build())
        .setHeader(HttpHeaders.CONTENT_TYPE,
            MediaType.JSON_UTF_8.toString())
        .setBodyGenerator(jsonBodyGenerator(taskUpdateRequestCodec,
            updateRequest))
        .build();

    ListenableFuture<JsonResponse<TaskInfo>> future =
        httpClient.executeAsync(request, createFullJsonResponseHandler(taskInfoCodec));
    .....
}
```

从上面的代码可以看出，需要将当前 Task 处理的数据都封装成 `TaskSource` 的列表，其中一个 `TaskSource` 代表一个 Task 处理的数据源，而 Task 处理的数据源又分为两类：Stage 的输出和直接的数据源。对于 Stage 输出类型的数据源，`TaskSource` 类封装了一个 Stage 的 `PlannodeId` 和根据该 Stage 上的 `TaskLocation` 生成的 `ScheduledSplit` 列表；对于直接的数据源，`TaskSource` 类封装了一个数据源的 `PlannodeId` 和根据该数据源上的所有数据分片生成的 `ScheduledSplit` 列表。从上面的代码可以看出，通过方法 `getSources()` 生成 `TaskSource` 列表，在该方法中会将两种数据源（Stage 的输出和直接的数据源）都封装成 `TaskSource`，然

后合并到一起。该方法的代码如下所示：

```
private synchronized List<TaskSource> getSources() {
    // planFragment.getPartitionedSourceNode() 返回的是直接数据源对应的 PlanNode 对象
    planFragment.getRemoteSourceNodes() 返回的是上游 Stage 对应的 PlanNode 对象
    return Stream.concat(Stream.of(planFragment.getPartitionedSourceNode()),
        planFragment.getRemoteSourceNodes().stream())
        .filter(Objects::nonNull)
        .map(PlanNode::getId)
        .map(this::getSource)
        .filter(Objects::nonNull)
        .collect(toImmutableList());
}
```

2. Resource 端

从上面内容可以看到，在 Presto 中对 Task 的启动其实是在 Coordinator 上根据数据本地性等条件，向特定的 Worker 节点上的 TaskResource 类提供的 RESTful 服务发送类似 /v1/task/taskId?summarize 的 POST 请求来实现的。通过发送此 RESTful 请求，在特定的 Worker 上启动一个 SqlTaskExecution 对象，用于执行 Task 计算任务。

在 Presto 集群启动的时候，会在每个 Worker 上都启动一个 PrestoServer 进程，该进程会提供 RESTful 服务，用于处理前缀为 /v1/task/ 的 URI 标示的 RESTful 请求。而对于形如 /v1/task/* 的 RESTful 请求的处理均由类 TaskResource 完成。TaskResource 类对创建 Task 的 RESTful 请求的处理函数如下：

```
@Path("/v1/task")
public class TaskResource
{
    .....
    @POST
    @Path("/{taskId}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response createOrUpdateTask(@PathParam("taskId") TaskId taskId,
        TaskUpdateRequest taskUpdateRequest, @Context UriInfo uriInfo)
    {
        //由于 TaskId 是新建的，因此创建并启动一个 Task
        TaskInfo taskInfo = taskManager.updateTask
(taskUpdateRequest.getSession(),
            taskId,
            taskUpdateRequest.getFragment(),
            taskUpdateRequest.getSources(),
            taskUpdateRequest.getOutputIds());
        if (shouldSummarize(uriInfo)) {
            taskInfo = taskInfo.summarize();
        }
        return Response.ok().entity(taskInfo).build();
    }
    .....
}
```


从上面的代码可以看出，对新建 Task 的 RESTful 的处理主要是由方法 `taskManager.updateTask(...)` 来完成的，其实就是调用类 `SqlTaskManager` 中的 `updateTask(...)` 方法。该方法的内容如下：

```
public TaskInfo updateTask(Session session, TaskId taskId, PlanFragment fragment,
    List<TaskSource> sources, OutputBuffers outputBuffers) {
    .....
    //tasks 是一个全局缓存，根据 taskId 获得已经缓存的 SqlTask，若没有与之对应的
    //SqlTask，则会新建一个
    SqlTask sqlTask = tasks.getUnchecked(taskId);
    sqlTask.recordHeartbeat();
    //执行 SqlTask，并返回执行信息
    return sqlTask.updateTask(session, fragment, sources, outputBuffers);
}
```

上面的代码是通过 `tasks.getUnchecked(taskId)` 获得 `SqlTask` 对象的，其实 `tasks` 是一个 `com.google.common.cache.LoadingCache` 对象，因此，若根据 `taskId` 第一次获得对应的 `SqlTask` 对象是需要创建该 `SqlTask` 对象的，而怎么创建 `SqlTask` 对象，是在 `LoadingCache` 对象初始化的时候就已经指定了的，代码如下：

```
tasks = CacheBuilder.newBuilder().build(new CacheLoader<TaskId, SqlTask>()
{
    @Override
    public SqlTask load(TaskId taskId) throws Exception {
        return new SqlTask(
            taskId,
            nodeInfo.getInstanceId(),
            locationFactory.createLocalTaskLocation(taskId),
            queryContexts.getUnchecked(taskId.getQueryId()),
            sqlTaskExecutionFactory,
            taskNotificationExecutor,
            sqlTask -> {
                finishedTaskStats.merge(sqlTask.getIoStats());
                return null;
            },
            maxBufferSize
        );
    }
});
```

从上面的代码可以看出，每次 `taskId` 对应的 `SqlTask` 对象不存在的时候，都会重新创建一个。

获得了 `SqlTask` 对象之后就会调用 `sqlTask.updateTask(session, fragment, sources, outputBuffers)` 方法启动 Task，进行计算，其代码如下：

```
public TaskInfo updateTask(Session session, PlanFragment fragment,
    List<TaskSource> sources, OutputBuffers outputBuffers)
{
    try {
        SqlTaskExecution taskExecution;
```



```

.....
//在新建的 SqlTask 对象中第一次执行 updateTask 方法的时候, taskExecution
为 null
taskExecution = taskHolder.getTaskExecution();
//若 taskExecution 为 null, 则创建一个
if (taskExecution == null) {
    taskExecution = sqlTaskExecutionFactory.create(session,
queryContext, taskStateMachine, sharedBuffer, fragment, sources);
    taskHolderReference.compareAndSet(taskHolder, new
TaskHolder(taskExecution));
}
if (taskExecution != null) {
    // 若不为 null, 则更新当前 SqlTask 的 OutputBuffers (sharedBuffer 用于输
出当前 Task 产生的结果), 并将新的 TaskSource 添加到当前 SqlTask 对应的 taskExecution 中
    sharedBuffer.setOutputBuffers(outputBuffers);
    //该方法将 sources 中包含的所有的 Split 都合并到 taskExecution 现在需要处理的
Split 列表中
    taskExecution.addSources(sources);
}
}
//返回当前 Task 的信息
return getTaskInfo();
}

```

从上面的代码可知, updateTask 方法的唯一作用就是更新 TaskId 对应的 Task 的 OutputBuffers 和需要处理的 Split 列表。

7.3.2 更新 Task

Coordinator_Only、Single 和 Fixed 类型的 Task 在调度的时候都是直接创建 Task 的, 而只有 Source 类型的 Task 由于其分批调度 Splits, 因此有可能会多次调用 scheduleTask 方法, 所以会判断在调用 scheduleTask 方法的时候, 对应的 Node 上是否已经创建了 Task; 若已经创建, 就更新 Task; 否则就创建 Task。因此在 Task 调度阶段只有 Source 类型的 Task 才会被更新。

1. Client 端

在 Client 端更新 Task 的过程分为两步: 根据 taskId 查找对应的 HttpRemoteTask 对象和调用创建 HttpRemoteTask 对象的 addSplits() 方法, 这两步操作均在 SqlStage Execution.assignSplits(AtomicInteger nextTaskId, Multimap<Node, Split> splitAssignment)方法中完成, 下面对该方法进行说明:

```

private void assignSplits(AtomicInteger nextTaskId, Multimap<Node, Split>
splitAssignment) {
    //遍历各个 Entry<Node, Collection<Split>>
    for (Entry<Node, Collection<Split>> taskSplits :
splitAssignment.asMap().entrySet()) {
        //找到各个 Entry<Node, Collection<Split>>中的 key, 一个 Entry 中的 key 是 Node,
value 是 Collection<Split>, 因此一个 Entry 表示存储在一个 Node 上的 Split 集合
        Node node = taskSplits.getKey();

```

```

        //根据 Node 获得运行在该 Node 上的 Task 的 taskId
        TaskId taskId = Iterables.getOnlyElement(localNodeTaskMap.get(node),
null);

        //根据 taskId 获得相应的 RemoteTask
        RemoteTask task = taskId != null ? tasks.get(taskId) : null;
        if (task == null) {
            //若 Task 为 null, 则新建一个 Task, 并按照 7.3.1 节所述进行后续处理
            .....
        }
        else {
            //若 Task 不为 null, 则将位于 Task 所在 Node 上的 Split 添加到该 Task 上
            task.addSplits(fragment.getPartitionedSource(),
taskSplits.getValue());
            .....
        }
    }
}

```

从上面的代码可以看出, 更新 Task 的 Client 端操作主要包括 `HttpRemoteTask` 对象的查找和 `addSplits()` 方法的调用, 下面针对这两步操作进行详细讲解。

(1) 查找 `HttpRemoteTask` 对象

查找 `HttpRemoteTask` 对象的步骤如下。

步骤 1: 每一批 Split 都位于一系列的 Node 中, 遍历每个 Node, 然后针对于每个 Node 执行步骤 2。

步骤 2: 根据每个 Node 获得运行在该 Node 上的 Source Task 的 `taskId`, 然后针对每个 `taskId` 执行步骤 3。

步骤 3: 根据每个 `taskId` 查找对应的 `HttpRemoteTask`。

通过上述 3 个步骤就可以找到相应的 `HttpRemoteTask`。找到了相应的 `HttpRemoteTask` 之后就需要调用该 `HttpRemoteTask` 对象上的方法 `addSplits(...)`。通过该方法可以向 `TaskResource` 端发起更新 Task 的请求。该方法的内容如下:

```

public synchronized void addSplits(PlanNodeId sourceId, Iterable<Split> splits)
{
    try (SetThreadName ignored = new SetThreadName("HttpRemoteTask-%s",
taskId)) {
        .....
        // 只有当前的 Task 还没有执行完毕, 才会进行 Split 追加
        if (!getTaskInfo().getState().isDone()) {
            int added = 0;
            for (Split split : splits) {
                if (pendingSplits.put(sourceId, new
ScheduledSplit(nextSplitId.getAndIncrement(), split))) {
                    added++;
                }
            }
            if (sourceId.equals(planFragment.getPartitionedSource())) {
                pendingSourceSplitCount += added;
            }
        }
    }
}

```

```

        needsUpdate.set(true);
    }
    //更新了 Split, 向 TaskResource 发起更新 Split 的请求
    scheduleUpdate();
}
}

```

通过上面的代码可以看到, 将 Split 添加到 HttpRemoteTask 中之后, 是通过调用 scheduleUpdate()方法, 向 Resource 发起 Split 更新请求的, 该方法的内容如下:

```

private synchronized void scheduleUpdate() {
    //若 Task 还未开始执行或者 Task 已经结束了, 则不需要做任何操作, 直接从方法返回
    if (!needsUpdate.get() || taskInfo.get().getState().isDone()) {
        return;
    }
    .....
    //将当前 Task 需要处理的所有 Split 都组装成 TaskSource 列表
    List<TaskSource> sources = getSources();
    //使用 TaskSource 列表、outputbuffers 创建 request 请求
    TaskUpdateRequest updateRequest = new TaskUpdateRequest(session,
        planFragment,
        sources,
        outputBuffers.get());
    //生成一个地址为: http://WorkerNodeIP:Port/v1/task/taskId?summarize 的 Post
    请求
    Request request = preparePost()
        .setUri(uriBuilderFrom(taskInfo.get().getSelf()).addParameter("s
ummarize").build())
        .setHeader(HttpHeaders.CONTENT_TYPE, MediaType.JSON_UTF_8.
toString())
        .setBodyGenerator(jsonBodyGenerator(taskUpdateRequestCodec,
updateRequest))
        .build();
    //发起请求
    ListenableFuture<JsonResponse<TaskInfo>> future =
httpClient.executeAsync(request, createFullJsonResponseHandler(taskInfoCodec));
    .....
}

```

从上面的代码可以看出, 需要将当前 Task 处理的数据都封装成 TaskSource 的列表, 其中一个 TaskSource 代表一个 Task 处理的数据源, 而 Task 处理的数据源又分为两类: Stage 的输出和直接的数据源。对于 Stage 输出类型的数据源, TaskSource 类封装了一个 Stage 的 PlanNodeId 和根据该 Stage 上的 TaskLocation 生成的 ScheduledSplit 列表; 对于直接的数据源, TaskSource 类封装了一个数据源的 PlanNodeId 和根据该数据源上的所有数据分片生成的 ScheduledSplit 列表。从上面的代码可以看出, 通过方法 getSources()生成 TaskSource 列表, 在该方法中会将两种数据源 (Stage 的输出和直接的数据源) 都封装成 TaskSource, 然后合并到一起。该方法的代码如下所示:


```

private synchronized List<TaskSource> getSources() {
    // planFragment.getPartitionedSourceNode() 返回的是直接数据源对应的 PlanNode 对象
    planFragment.getRemoteSourceNodes() 返回的是上游 Stage 对应的 PlanNode 对象
    return Stream.concat(Stream.of(planFragment.getPartitionedSourceNode()),
        planFragment.getRemoteSourceNodes().stream())
        .filter(Objects::nonNull)
        .map(PlanNode::getId)
        .map(this::getSource)
        .filter(Objects::nonNull)
        .collect(toImmutableList());
}

```

2. Resource 端

由于 Task 的更新操作主要发生在 Source Stage，因此，在 Presto 中，对 Task 的更新其实是在 Coordinator 上根据数据本地性等条件，查找已经运行的所有的 Task，向特定的 Worker 节点上的 TaskResource 类提供的 RESTful 服务发送类似/v1/task/taskId?summarize 的 POST 请求，从而更新运行在该 Worker 上的 Source Task 需要处理的数据。

在 Presto 集群启动的时候，会在每个 Worker 上都启动一个 PrestoServer 进程，该进程会提供 RESTful 服务，用于处理前缀为/v1/task/的 URI 标示的 RESTful 请求。而针对于/v1/task/* 类型 RESTful 请求的处理均由类 TaskResource 完成。TaskResource 类对创建 Task 的 RESTful 请求的处理函数如下：

```

@Path("/v1/task")
public class TaskResource
{
    .....
    @POST
    @Path("/{taskId}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response createOrUpdateTask(@PathParam("taskId") TaskId taskId,
        TaskUpdateRequest taskUpdateRequest, @Context UriInfo uriInfo)
    {
        //与创建 Task 不同，由于该 Node 上已经运行了 Task，因此可以通过传入的 TaskId 找到相应的
        Task
        TaskInfo taskInfo =
            taskManager.updateTask(taskUpdateRequest.getSession(),
                taskId,
                taskUpdateRequest.getFragment(),
                taskUpdateRequest.getSources(),
                taskUpdateRequest.getOutputIds());
        if (shouldSummarize(uriInfo)) {
            taskInfo = taskInfo.summarize();
        }
        return Response.ok().entity(taskInfo).build();
    }
    .....
}

```


从上面的代码可以看出, RESTful 服务对更新 Task 的请求的处理主要是由方法: `taskManager.updateTask(...)` 来完成的, 其实就是调用类 `SqlTaskManager` 中的 `updateTask(...)` 方法。该方法的内容如下:

```
public TaskInfo updateTask(Session session, TaskId taskId, PlanFragment fragment,
List<TaskSource> sources, OutputBuffers outputBuffers) {
    .....
    //tasks 是一个全局缓存, 根据 taskId 获得已经缓存的 SqlTask, 若没有与之对应的 SqlTask,
    则会新建一个新的。由于 Task 已经运行在该节点上了, 因此肯定可以根据 taskId 找到相应的 Task
    SqlTask sqlTask = tasks.getUnchecked(taskId);
    sqlTask.recordHeartbeat();
    //执行 SqlTask, 并返回执行信息
    return sqlTask.updateTask(session, fragment, sources, outputBuffers);
}
```

从上面的代码可以看出, 是通过 `tasks.getUnchecked(taskId)` 获得 `SqlTask` 对象的, 其实 `tasks` 是一个 `com.google.common.cache.LoadingCache` 对象, 因此, 若根据 `taskId` 第一次获得对应的 `SqlTask` 对象是需要创建该 `SqlTask` 对象的, 具体怎么创建对象在前面已经描述了, 在此就不再赘述。由于是更新 Task, 因此一定可以根据 `taskId` 找到对应的 `SqlTask` 对象。

获得了 `SqlTask` 对象之后就会调用 `sqlTask.updateTask(session, fragment, sources, outputBuffers)` 方法更新 Task, 其代码如下:

```
public TaskInfo updateTask(Session session, PlanFragment fragment,
List<TaskSource> sources, OutputBuffers outputBuffers)
{
    try {
        SqlTaskExecution taskExecution;
        .....
        //在新建的 SqlTask 对象中第一次执行 updateTask 方法的时候, taskExecution
        为 null; 若不是第一次调用, 那么 taskExecution 就不为 null。
        taskExecution = taskHolder.getTaskExecution();
        //若 taskExecution 为 null, 则创建一个
        if (taskExecution == null) {
            taskExecution = sqlTaskExecutionFactory.create(session,
queryContext, taskStateMachine, sharedBuffer, fragment, sources);
            taskHolderReference.compareAndSet(taskHolder, new
TaskHolder(taskExecution));
        }
        if (taskExecution != null) {
            // 若不为 null, 则更新当前 SqlTask 的 OutputBuffers (sharedBuffer 用于输
            出当前 Task 产生的结果), 并将新的 TaskSource 添加到当前 SqlTask 对应的 taskExecution 中
            sharedBuffer.setOutputBuffers(outputBuffers);
            //该方法将 sources 中包含的所有 Split 都合并到 taskExecution 现在需要处理的
            Split 列表中
            taskExecution.addSources(sources);
        }
    }
    //返回当前 Task 的信息
    return getTaskInfo();
}
```

从上面的代码可知，`updateTask` 方法的唯一作用就是更新 `TaskId` 对应的 `Task` 的 `OutputBuffers` 和需要处理的 `Split` 列表。

7.3.3 运行 Task

通过前面的章节我们可以看到，在 Presto 中，每个 `Task` 是通过 `HttpRemoteTask` 类中的 `HttpClient` 向 `TaskResource` 提交 `Task` 创建或更新请求的，每个 `Task` 在 `TaskResource(Worker)` 端最终是以 `SqlTaskExecution` 运行的，本节会在 `TaskResource` 端针对 `Task` 的运行进行详细说明。

1. TaskExecutor

要想了解 `Task` 的运行过程，首选需要了解一个类：`TaskExecutor`。`TaskExecutor` 是运行在 Presto 集群的每个 `Worker` 上的 Presto 服务的，它是用于运行实际计算任务的线程池的包装类，该类的主要作用就是处理在 `Worker` 上运行的所有 `Task` 中的 `Split`。

`TaskExecutor` 通过 Guice 依赖注入到 `SqlTaskManager` 中，并在每个 `Worker` 启动的时候调用其 `start()` 方法运行，其 `start()` 方法的内容如下：

```
// PostConstruct 注解说明：start() 方法会在 TaskExecutor 类构造方法调用完毕之后直接被调用
@PostConstruct
public synchronized void start()
{
    // runnerThreads 的值是通过配置参数：task.max-worker-threads 进行配置的，默认值为当前
    CPU 核数*4
    for (int i = 0; i < runnerThreads; i++) {
        //启动一个新的 Split 处理线程，并加入到线程池
        addRunnerThread();
    }
}
//启动一个线程，并加入到线程池，Runner 是 TaskExecutor 的内部类
private synchronized void addRunnerThread() {
    try {
        //Runner 是内部类：com.facebook.presto.execution.TaskExecutor.Runner
        executor.execute(new Runner());
    }
    catch (RejectedExecutionException ignored) {
    }
}
```

7.3.3.2 TaskExecutor.Runner

上面的代码在 `TaskExecutor` 类中启动多个线程，并将各个线程加入到线程池中。对各个 `Split` 的处理都是由 `Runner` 类来完成的，该类为 `TaskExecutor` 的内部类，其对 `Split` 处理的代码如下：

```
private class Runner implements Runnable {
    .....
    @Override
    public void run() {
```

```

try (SetThreadName runnerName = new SetThreadName("SplitRunner-%s", runnerId))
{
    //只要当前 TaskExecutor 不结束并且当前线程不被中断, 当前线程就一直不停地循环获取
    pendingSplits 中的 Split 进行处理
    while (!closed && !Thread.currentThread().isInterrupted()) {
        //获得下一个需要处理的 PrioritizedSplitRunner 对象, PrioritizedSplitRunner 是对作
        用于一个 Split 所有操作的包装, 封装了作用于一个 Split 上的一系列的 Operator
        final PrioritizedSplitRunner split;
        try {
            split = pendingSplits.take();
            .....
            try (SetThreadName splitName = new
SetThreadName(split.getTaskHandle().getTaskId() + "-" + split.getSplitId())) {
                //将取出的 Split 加入到 runningSplits 队列, 该队列中保存了所有正在处理的 Split
                runningSplits.add(split);
                boolean finished;
                ListenableFuture<?> blocked;
                try {
                    //调用各个 Split 的 process() 方法, process() 方法只会执行固定时间片长度, 若过了
                    固定的时间片 Split 还没有处理完毕, 则也会返回
                    blocked = split.process();
                    finished = split.isFinished();
                }
                finally {
                    //执行完毕之后, 需要将 Split 从 runningSplits 中移除
                    runningSplits.remove(split);
                }
                //注意: finished 表示整个 Split 是否已经处理完毕, blocked.isDone() 表示本次执
                行是否完毕
                if (finished) {
                    log.debug("%s is finished", split.getInfo());
                    splitFinished(split);
                }
                else {
                    //若本次执行完毕, 则放入到 pendingSplits 队列中
                    if (blocked.isDone()) {
                        pendingSplits.put(split);
                    }
                    //若本次还未执行完毕, 则放入到 blockedSplits 中, 并添加执行完毕时的
                    listener
                    else {
                        blockedSplits.put(split, blocked);
                        blocked.addListener(new Runnable()
                        {
                            @Override
                            public void run()
                            {
                                //一旦固定时间片执行完毕, 则从 blockedSplits 中移除,
                                并加入到 pendingSplits 中
                                blockedSplits.remove(split);
                                split.updatePriorityLevel();
                                pendingSplits.put(split);
                            }
                        });
                    }
                }
            }
        }
    }
}

```



```

        }
        }, executor);
    }
}
}
catch (Throwable t) {
    if (t instanceof PrestoException) {
        PrestoException e = (PrestoException) t;
        log.error("Error processing %s: %s: %s", split.getInfo(),
e.getErrorCode().getName(), e.getMessage());
    }
    else {
        log.error(t, "Error processing %s", split.getInfo());
    }
    splitFinished(split);
}
}
}
finally {
    // 若代码执行到这里, 说明上面的循环结束, 即: TaskExecutor..Runner 线程被中
    断或 TaskExecutor 结束。这时首先判断上述循环结束的原因: TaskExecutor.Runner 线程被中断但是
    TaskExecutor 尚未结束, 则需要添加一个新的 Runner 继续执行, 从而保证 TaskExecutor 线程池中时始终
    保持固定数目的处理线程。
    if (!closed) {
        //closed 标示 TaskExecutor 是否结束
        addRunnerThread();
    }
}
}
}
}
}

```

3. PrioritizedSplitRunner

从上面的代码可以看出, 所有对 Split 的处理均由 split.process() 完成, 因此主要看 split.process() 方法的内容。此处的 Split 是 PrioritizedSplitRunner 对象, PrioritizedSplitRunner.process() 方法的内容如下:

```

public ListenableFuture<?> process() throws Exception {
    try {
        .....
        //调用 split 的 processFor(Duration duration) 方法进行实际的 split 的处理
        ListenableFuture<?> blocked = split.processFor(SPLIT_RUN_QUANTA);
        .....
        return blocked;
    }
    catch (Throwable e) {
        .....
    }
}
}

```

4. DriverSplitRunner

从上述代码可以看出, Split 的处理由方法 split.processFor(SPLIT_RUN_QUANTA) 完成,

其中 Split 是类 PrioritizedSplitRunner 中的某个属性的属性名，其类型为 SplitRunner，而 SplitRunner 其实只是一个接口，在 Presto runtime 中的实现类为 DriverSplitRunner。其中 processFor 方法的传入参数 SPLIT_RUN_QUANTA 指的是时间段，该值默认为 1 秒，表示该方法最多只能运行 1 秒，若 1 秒之后 Split 还没有处理完毕，则会直接返回相应的 ListenableFuture，然后由 TaskExecutor.Runner 线程继续获得 pendingSplits 中的下一个 Split 进行处理。关于 DriverSplitRunner.processFor(SPLIT_RUN_QUANTA)方法的代码如下：

```
@Override
public ListenableFuture<?> processFor(Duration duration) {
    Driver driver;
    synchronized (this) {
        // 若在执行该方法之前 DriverSplitRunner 就已经结束了，那么就没有必要再进行后续操作了，直接
        // 返回一个 value 为 null 的 ListenableFuture 即可
        if (closed) {
            return Futures.immediateFuture(null);
        }
        // 若当前的 Driver 为 null，则需要首先根据 Client 指定的 Split 创建一个 driver，
        // partitionedSplit 是类 DriverSplitRunner 中的属性，其类型为 ScheduledSplit，而 ScheduledSplit
        // 是 Split 的封装类
        if (this.driver == null) {
            this.driver = driverSplitRunnerFactory.createDriver(driverContext,
                partitionedSplit);
        }
        driver = this.driver;
    }
    // driver 是作用于 split 上的一系列的 operator 的封装类，driver 需要处理的 Split 存储在属性
    // newSources 中
    return driver.processFor(duration);
}
```

5. Driver

从上面的代码可以看出，作用于 Split 上的一系列操作的封装类为 Driver 类，其实例为 DriverSplitRunner 中的属性 driver，因此对 Split 的处理操作集中在 driver 中，下面对 driver.processFor(duration)方法进行说明。

```
public ListenableFuture<?> processFor(Duration duration)
{
    // 计算最多可以运行的时间
    long maxRuntime = duration.roundTo(TimeUnit.NANOSECONDS);
    // 当前线程获得锁。若有其他线程持有锁，则最多等待 100 毫秒。若超过 100 毫秒还没有获得锁，
    // 则 lockResult.wasAcquired() 的值为 false
    try (DriverLockResult lockResult = tryLockAndProcessPendingStateChanges
        (100, TimeUnit.MILLISECONDS)) {
        if (lockResult.wasAcquired()) {
            driverContext.startProcessTimer();
            try {
                long start = System.nanoTime();
                do {
                    // 由 processInternal() 方法完成实际的 Split 处理

```

```

        ListenableFuture<?> future = processInternal();
        if (!future.isDone()) {
            return future;
        }
    }
    while (System.nanoTime() - start < maxRuntime
    && !isFinishedInternal());
    }
    finally {
        driverContext.recordProcessed();
    }
    }
    //若未获得锁，则直接返回 NOT_BLOCKED
    return NOT_BLOCKED;
}

```

从上面的代码可以看出，对 Split 的实际处理工作其实是由方法 `Driver.processInternal()` 完成的。该方法的实现代码如下：

```

private ListenableFuture<?> processInternal()
{
    try {
        //对于处于 SourceStage 的 Task，若尚有未处理读取的 Split，将未读取的 Split 加入
        到 SourceOperator 中
        if (!newSources.isEmpty()) {
            processNewSources();
        }
        //若只有一个 operator，则需要进行单独处理
        if (operators.size() == 1) {
            //若当前 Driver 已经执行完毕，则直接返回 NOT_BLOCKED
            if (driverContext.isDone()) {
                return NOT_BLOCKED;
            }
            //获得 operator
            Operator current = operators.get(0);
            //通过方法 isBlocked(Operator o) 返回指定的 operator 是否阻塞
            ListenableFuture<?> blocked = isBlocked(current);
            //若阻塞，则直接返回 blocked 所指向的 ListenableFuture
            if (!blocked.isDone()) {
                current.getOperatorContext().recordBlocked(blocked);
                return blocked;
            }
            // 若未阻塞，则直接结束当前 Operator
            current.getOperatorContext().startIntervalTimer();
            current.finish();
            current.getOperatorContext().recordFinish();
            return NOT_BLOCKED;
        }
    }

    boolean movedPage = false;
    //若 operators 的个数大于 1，则执行下面的循环，从下面的循环可以看出：每次从

```

operators 中取出相邻的两个 operator，得到前一个 operator 的输出数据，然后将该输出数据作为后一个 operator 的输入数据。

```

for (int i = 0; i < operators.size() - 1 && !driverContext.isDone();
i++) {
    //一次取出相邻的两个 operator
    Operator current = operators.get(i);
    Operator next = operators.get(i + 1);
    // 跳过已经执行结束的 operator
    if (!isBlocked(current).isDone()) {
        continue;
    }
    if (!isBlocked(next).isDone()) {
        continue;
    }

    // 如果当前 operator 没有结束，而且下一个 operator 也需要输入
    if (!current.isFinished() && next.needsInput()) {
        //从当前 operator 中获得 output page，然后将该 page 作为输入，交给下一个
operator 进行处理

        current.getOperatorContext().startIntervalTimer();
        Page page = current.getOutput();
        current.getOperatorContext().recordGetOutput(page);
        // 将获得的 output page 交给下一个 operator 进行处理
        if (page != null) {
            next.getOperatorContext().startIntervalTimer();
            next.addInput(page);
            next.getOperatorContext().recordAddInput(page);
            //标示：已经进行了 Page 的移动
            movedPage = true;
        }
    }

    //若当前的 operator 已经完成了，则通知下一个 operator：不会再有输入了，需要
完成数据处理，并将结果进行刷新
    if (current.isFinished()) {
        next.getOperatorContext().startIntervalTimer();
        next.finish();
        next.getOperatorContext().recordFinish();
    }
}

// 若所有的 operator 都已经循环完毕了，但是没有发生 Page 的移动，我们需要检查是否
有 operator 被 block 住了
if (!movedPage) {
    List<Operator> blockedOperators = new ArrayList<>();
    List<ListenableFuture<?>> blockedFutures = new ArrayList<>();
    //循环所有的 operator，并获得每个 operator 的 ListenableFuture 对象，
isBlocked() 方法会进行判断：若当前 operator 已经执行结束，则会返回其是否在等待额外的内存
    for (Operator operator : operators) {
        ListenableFuture<?> blocked = isBlocked(operator);
        if (!blocked.isDone()) {
            blockedOperators.add(operator);
            blockedFutures.add(blocked);
        }
    }
}

```



```

    }

    //若确实有 operator 被阻塞住了
    if (!blockedFutures.isEmpty()) {
        //任意一个 ListenableFuture 完成, 就会解除当前 Driver 的阻塞状态
        ListenableFuture<?> blocked =
firstFinishedFuture(blockedFutures);
        // 当前 Driver 添加 monitor 实时监听是否已经解除阻塞状态
driverContext.recordBlocked(blocked);
        // 为每个 operator 都注册已经监听器, 实时监听是否已经解除阻塞状态
        for (Operator operator : blockedOperators) {
            operator.getOperatorContext().recordBlocked(blocked);
        }
        return blocked;
    }

    return NOT_BLOCKED;
}

catch (Throwable t) {
    driverContext.failed(t);
    throw t;
}

}

//该方法被多次调用, 判断指定的 operator 是否阻塞
private static ListenableFuture<?> isBlocked(Operator operator)
{
    ListenableFuture<?> blocked = operator.isBlocked();
    //若 operator 已经执行结束, 还需要判断当前 operator 是否在等待额外的内存
    if (blocked.isDone()) {
        blocked = operator.getOperatorContext().isWaitingForMemory();
    }
    return blocked;
}

//该方法用来为所有的 Future 添加一个解锁器, 有任意一个 ListenableFuture 完成, 则会
解除阻塞状态
private static ListenableFuture<?>
firstFinishedFuture(List<ListenableFuture<?>> futures) {
    SettableFuture<?> result = SettableFuture.create();
    ExecutorService executor = MoreExecutors.newDirectExecutorService();

    for (ListenableFuture<?> future : futures) {
        future.addListener(() -> result.set(null), executor);
    }

    return result;
}

```

通过上面的代码我们可以看出: 每个 Driver 中封装了对 Split 的所有操作, 每次执行 Split 计算的时候, 我们都会依次遍历作用于该 Split 上的所有 operator, 取出当前 operator 的 output page, 然后将该 output page 作为下一个 operator 的 input page, 交给下一个 operator 进行处

理。直到将该 Driver 封装的所有 Operator 遍历完毕。

Operator 中对 page 的处理集中在两个方法中：`addInput(Page page)`和 `getOutput()`，由于 Operator 是一个接口，其实现类有很多，不同的实现类对 page 的处理方式都不一样，因此可能会在 `addInput(Page page)`方法中完成对 Page 的计算和处理，也可能在 `getOutput()`方法中完成对 Page 的计算和处理，还有可能两个方法结合起来完成对 page 的计算和处理。

7.4 小结

本章对 Presto 的查询执行过程进行了深入细致的讲解，并分别从查询执行逻辑、Task 生成、Task 调度和 Task 执行等多个方面对 Presto 中的查询执行进行了全面、深入代码的细致研究和说明，通过本章，读者可以一探 Presto 的核心：查询执行，从而为后续对 Presto 的高阶理解和应用打下坚实的基础。

第 8 章

队 列

通过第 4 章我们得知，StatementResource 接收到查询请求后，通过 com.facebook.presto.execution.SqlQueryManager.createQuery 方法创建查询，并通过 com.facebook.presto.execution.SqlQueryQueueManager.submit 方法来提交并启动查询，提交查询时需要满足 Presto 提供的队列规则才可以启动查询。

Presto 队列是用于控制查询并发量和可接收的 SQL 数量，可针对用户、提交来源、Session 等信息进行个性化配置。队列规则定义在一个 JSON 文件内，可在 \$PRESTO_HOME/etc/config.properties 中通过配置参数 query.queue-config-file 来指定队列规则的 JSON 文件路径。

8.1 配置说明

Presto 的队列规则配置很简单，例如在 \$PRESTO_HOME/etc/目录下新增 queues.json 文件，用于配置所有的队列规则信息，并设置 query.queue-config-file 参数指向到该文件的所在路径，例如该参数的配置如下：

```
query.queue-config-file=etc/queues.json
```

可依据如下样例，在 queues.json 配置详细的队列规则信息：

```
{
  "queues": {
    "user.a": {
      "maxConcurrent": 5,
      "maxQueued": 20
    },
    "user.b": {
      "maxConcurrent": 6,
      "maxQueued": 10
    },
    "global": {
      "maxConcurrent": 20,
```

```

        "maxQueued": 1000
    },
    "rules": [
        {
            "user": "a",
            "source": ".*presto-cli*",
            "session.experimental_big_query": "true",
            "queues": [
                "user.a",
                "global"
            ],
        },
        {
            "user": "b",
            "queues": [
                "user.b",
                "global"
            ]
        }
    ]
}

```

从以上的配置样例可以看出，Presto 的队列规则信息分成两大部分：queues 和 rules。

8.1.1 queues 队列定义

从上述得知队列规则配置文件中包括 queues 和 rules 两部分，本节会针对这两部分的内容进行详细说明以及介绍如何进行队列规则的配置。

队列信息定义在 queues 标签下，每个队列包含 3 个属性：

- queueName: 队列名称，也就是每个队列的标签名称，例如 user.a, user.b 等，是每个队列 json 对的 key 值。
- maxConcurrent: 该队列允许同时运行查询的最大并发数量。
- maxQueued: 该队列允许同时接收提交查询请求的最大数量。

例如定义一个 global 的队列，配置的样例如下：

```

"global": {
    "maxConcurrent": 20,
    "maxQueued": 1000
}

```

其中，global 为 queueName，20 和 1000 分别为 maxConcurrent 和 maxQueued 的值。

8.1.2 rules 规则定义

规则定义在 rules 标签下，每个规则可定义 4 个属性，每个属性的具体含义如下。

- user: 用户名，取值来源于 session 中的 user，若在 cli 中默认为操作系统用户；Presto JDBC 为传入的用户名。
- source: SQL 来源，目前一般有两种来源，即 presto-cli 和 presto-jdbc，其内容支持正

则表达式表示。

- session: session 参数定义, 范围在 System session 中所包含的参数。
- queues: 队列列表, 可以定义多个。

以下为配置样例:

```
{
  "user": "a",
  "source": ".*presto-cli.*",
  "session.experimental_big_query": "true",
  "queues": [
    "user.a",
    "global"
  ],
}
```

一个提交查询请求在匹配队列时, 按照以下的逻辑依次判断:

- 将查询请求中的 user 信息与规则中的 user 进行匹配, 若匹配成功, 则继续与查询请求中的 source 的内容继续匹配; 如果匹配成功, 则继续匹配 session 所定义的规则; 如果 session 定义的规则也满足, 则返回该规则下所定义的队列列表。
- 若 user 信息未匹配成功, 则继续与查询请求中 source 的内容继续匹配; 如果匹配成功, 则继续匹配 session 所定义的规则; 如果 session 定义的规则也满足, 则返回该规则下所定义的队列列表。
- 如果 user 和 source 信息均未匹配成功, 则匹配 session 所定义的规则; 如果满足, 则返回该规则下所定义的队列列表。
- 最终一个查询请求可能会匹配到多个规则, 默认取第一个。
- 最终所匹配到的规则中可能配置了多个队列, 如果都满足条件, 则默认选择第一个队列, 队列的匹配过程详见 8.3 节。

8.2 队列加载

Presto 队列信息的加载是在 `com.facebook.presto.execution.QueryQueueManager` 的实现类 `com.facebook.presto.execution.SqlQueryQueueManager` 中完成的, 两者关系在 `CoordinatorModule` 中进行绑定, 代码如下:

```
binder.bind(QueryQueueManager.class).to(SqlQueryQueueManager.class).in(Scopes.SINGLETON);
```

而队列的加载是在 `SqlQueryQueueManager` 的构造方法中完成的。

如果未配置队列的配置文件, 那么系统会自动构建两个队列: `global` 和 `big`。 `big` 队列对 session 参数 `experimental_big_query` 为 `true` 的查询定义了限制规则, 该队列限制查询的最大运行并发度为 10, 最大排队并发度为 500。不过这个 session 参数后续版本将不再启动; `global` 队列为其他的查询定义了限制规则, 该队列限制查询的最大运行并发度为 1000, 最大排队并发度为 5000。这部分的逻辑如以下代码所示:


```

ImmutableList.Builder<QueryQueueRule> rules = ImmutableList.builder();
if (config.getQueueConfigFile() != null) {
    // global 队列执行并发的最大值为 1000; 排队的最大值为 5000
    QueryQueueDefinition global = new QueryQueueDefinition("global",
config.getMaxConcurrentQueries(), config.getMaxQueuedQueries());
    // big 队列的执行并发的最大值为 10; 排队的最大值为 500
    QueryQueueDefinition big = new QueryQueueDefinition("big",
config.getMaxConcurrentBigQueries(), config.getMaxQueuedBigQueries());
    // 设置 big 队列规则, 将满足 session 参数 experimental_big_query 为 true 的查
    询
    rules.add(new QueryQueueRule(null, null, ImmutableMap.of(BIG_QUERY,
Pattern.compile("true", Pattern.CASE_INSENSITIVE)), ImmutableList.of(big)));
    // 设置 global 队列规则
    rules.add(new QueryQueueRule(null, null, ImmutableMap.of(),
ImmutableList.of(global)));
}

```

若配置了队列配置文件, 则依次读取队列和规则信息, 并将规则信息组装进 QueryQueueRule 列表中, 代码如下:

```

File file = new File(config.getQueueConfigFile());
ManagerSpec managerSpec;
try {
    managerSpec = mapper.readValue(file, ManagerSpec.class);
}
catch (IOException e) {
    throw Throwables.propagate(e);
}
Map<String, QueryQueueDefinition> definitions = new HashMap<>();
// 加载队列信息
for (Map.Entry<String, QueueSpec> queue :
managerSpec.getQueues().entrySet()) {
    definitions.put(queue.getKey(), new
QueryQueueDefinition(queue.getKey(), queue.getValue().getMaxConcurrent(),
queue.getValue().getMaxQueued()));
}
// 加载规则信息
for (RuleSpec rule : managerSpec.getRules()) {
    rules.add(QueryQueueRule.createRule(rule.getUserRegex(),
rule.getSourceRegex(), rule.getSessionPropertyRegexes(), rule.getQueues(),
definitions));
}

```

8.3 队列匹配

在 com.facebook.presto.execution.SqlQueryQueueManager.submit 方法内执行队列的匹配, 获取到查询请求的 session 信息后, 匹配规则信息, 若满足某一队列, 则返回当前队列中定义的队列。

```

private List<QueryQueue> selectQueues(Session session, Executor executor)
{

```

```

        for (QueryQueueRule rule : rules) {
            List<QueryQueueDefinition> definitions = rule.match(session);
            if (definitions != null) {
                return getOrCreateQueues(session, executor, definitions);
            }
        }
        throw new PrestoException(USER_ERROR, "Query did not match any queuing
rule");
    }
}

```

需要注意的是 `getOrCreateQueues` 方法，此方法将获取或者创建由 `com.facebook.presto.execution.QueryQueue` 对象组成的列表，`QueryQueue` 负责异步提交查询并维护该队列的容量值。`SqlQueryQueueManager` 维护了一个以 `QueueKey` 为 key，`QueryQueue` 为 value 的 map 对象：`queryQueues` 在 `getOrCreateQueues` 方法中，若未找到与 key 对应的 `QueryQueue` 对象，则创建一个新的 `QueryQueue` 并加入到该 map 中，代码如下：

```

private List<QueryQueue> getOrCreateQueues(Session session, Executor executor,
List<QueryQueueDefinition> definitions)
{
    ImmutableList.Builder<QueryQueue> queues = ImmutableList.builder();
    for (QueryQueueDefinition definition : definitions) {
        String expandedName = definition.getExpandedTemplate(session);
        QueueKey key = new QueueKey(definition, expandedName);
        if (!queryQueues.containsKey(key)) {
            QueryQueue queue = new QueryQueue(executor,
definition.getMaxQueued(), definition.getMaxConcurrent());
            if (queryQueues.putIfAbsent(key, queue) == null) {
                // Export the mbean, after checking for races
                String objectName = ObjectNames.builder(QueryQueue.class,
definition.getTemplate()).withProperty("expansion", expandedName).build();
                mbeanExporter.export(objectName, queue);
            }
        }
        queues.add(queryQueues.get(key));
    }
    return queues.build();
}

```

规则的匹配过程则简单得多，规则的匹配过程通过 `QueryQueueRule.match(Session session)` 完成，根据查询请求中的 `session`，按照在 8.1.2 节中所描述的匹配规则依次判断，实现过程如以下代码所示：

```

public List<QueryQueueDefinition> match(Session session)
{
    // 如果当前 rule 中定义了 user 信息，但是与查询请求中的 user 信息不匹配，则跳过该条 rule
    if (userRegex != null && !userRegex.matcher(session.getUser()).matches())
    {
        return null;
    }
    // 如果当前 rule 中定义了 source 信息，但是与查询请求中的 source 信息不匹配，则跳过该
条 rule
}

```

```

        if (sourceRegex != null) {
            String source = session.getSource();
            if (!sourceRegex.matcher(nullToEmpty(source)).matches()) {
                return null;
            }
        }
        // 如果当前 rule 中定义了 session 信息, 但与查询请求中的 session 均不匹配, 则跳过该条 rule
        for (Map.Entry<String, Pattern> entry : sessionPropertyRegexes.entrySet())
        {
            String value =
            session.getSystemProperties().getOrDefault(entry.getKey(), "");
            if (!entry.getValue().matcher(value).matches()) {
                return null;
            }
        }

        return queues;
    }

```

`selectQueues` 方法返回 `QueryQueue` 集合在选择具体由哪个 `QueryQueue` 执行查询前, 需要检查当前 `QueryQueue` 集合中的队列已执行和排队的查询数目是否已经达到队列的 `maxConcurrent` 和 `maxQueued` 总和上限值, 只要 `QueryQueue` 集合中有一个队列的已执行和排队的查询数目达到 `maxConcurrent` 和 `maxQueued` 总和的上限值, 则本次查询请求提交失败; 若都满足该要求, 选择 `QueryQueue` 集合的第一个队列提交查询, 提交前需要判断该队列已接收的查询请求数量是否已经达到当前运行查询数量和已排队的查询数量上限值之和, 若达到, 那么即使提交也会失败。这部分执行逻辑如下代码所示:

```

        for (QueryQueue queue : queues) {
            if (!queue.reserve(queryExecution)) {
                // Reject query if we couldn't acquire a permit to enter the queue.
                // The permits will be released when this query fails.
                return false;
            }
        }

        return queues.get(0).enqueue(createQueuedExecution(queryExecution,
        queues.subList(1, queues.size()), executor, stats));

```

其中 `reserve` 方法用于判断不能超过队列 `maxConcurrent` 和 `maxQueued` 的总和:

```

    public boolean reserve(QueryExecution queryExecution)
    {
        // queuePermits 等于 maxConcurrent 加 maxQueued 的值
        if (queuePermits.getAndDecrement() < 0) {
            queuePermits.incrementAndGet();
            return false;
        }

        addCompletionCallback(queryExecution, queuePermits::incrementAndGet);
        return true;
    }

```


`QueryQueue.enqueue` 方法用于判断不能超过队列 `maxQueuedQueries` 的值:

```
public boolean enqueue(QueuedExecution queuedExecution)
{
    if (queryQueueSize.incrementAndGet() > maxQueuedQueries) {
        queryQueueSize.decrementAndGet();
        return false;
    }
    .....
    // 异步提交查询请求
    asyncSemaphore.submit(entry);
    return true;
}
```

8.4 小结

本章从队列的配置及其实现讲述了 Presto 队列的具体用法及含义。当队列规则配置了多个队列时, 需要注意各个队列的最高并发度以及最大的排队数。Presto 队列的配置从执行并发度以及可排队的数量上进行了控制, 当集群规模发生变化或者数据源的数据量发生了量级的变化时, 需要考虑队列配置的调整, 以便使集群拥有最佳的计算性能。

第 9 章

System Connector

System Connector 提供了 Presto 集群的运行指标以及相关的元数据信息，这些集群信息也可以通过标准的 SQL 语句进行查询。System Connector 不需要配置专门的 System Catalog，Presto Server 启动后，System Connector 默认可用。

9.1 System Connector 使用

使用 `presto-cli` 进入到命令行后，用 `show schemas from system` 显示出 System Catalog 所提供的信息，如以下代码所示。

```
presto:default> show schemas from system;
      Schema
-----
information_schema
metadata
runtime
(3 rows)
```

9.1.1 Information_schema

Information_schema 提供的是 Presto Catalog 元数据信息，在默认情况下，每个 Presto Catalog 都会自动生成一个 `information_schema`。Information_schema 包含的信息有当前 Catalog 的 Schema 信息、Function 信息、Catalog 下所有的表信息以及表对应的列等，如下代码所示：

```
presto:default> use system.information_schema;
presto:information_schema> show tables;
      Table
-----
__internal_functions__
__internal_partitions__
```

```
columns
schemata
tables
views
(6 rows)
```

针对上面的输出，对每个表的含义进行说明如下。

- `__internal_functions__`：通过该表可以查询出 Presto 内置的 Functions 信息以及当前 Catalog 中自定义注册的 Functions。
- `__internal_partitions__`：通过该表可以查询出当前 Catalog 中包含的表的 Partitions 信息。
- `columns`：通过该表可以查询出当前 Catalog 中的列信息。
- `schemata`：通过该表可以查询出当前 Catalog 中的 Schema 信息。
- `tables`：通过该表可以查询出当前 Catalog 中的表信息。
- `views`：通过该表可以查询出当前 Catalog 中的视图信息。

以下为在 System Connector 中对 `Information_schema` 所包含的表的使用示例。

- 查询 System Catalog 内置的表信息。

```
presto:information_schema> select * from tables;
```

table_catalog	table_schema	table_name	table_type
system	information_schema	columns	BASE TABLE
system	information_schema	tables	BASE TABLE
system	information_schema	views	BASE TABLE
system	information_schema	schemata	BASE TABLE
system	information_schema	__internal_functions__	BASE TABLE
system	information_schema	__internal_partitions__	BASE TABLE
system	runtime	nodes	BASE TABLE
system	metadata	catalogs	BASE TABLE
system	runtime	queries	BASE TABLE
system	runtime	tasks	BASE TABLE

(10 rows)

- 查询 `system.runtime.nodes` 表的列信息。

```
presto:information_schema> select * from columns where table_name = 'nodes';
```

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	is_partition_key	comment
system	runtime	nodes	node_id	1		YES	varchar	NO	NULL
system	runtime	nodes	http_uri	2		YES	varchar	NO	NULL
system	runtime	nodes	node_version	3		YES	varchar	NO	NULL
system	runtime	nodes	coordinator	4		YES	boolean	NO	NULL
system	runtime	nodes	active	5		YES	boolean	NO	NULL

(5 rows)

- 查询 Presto 内置的 Functions 信息。

```
presto:information_schema> select * from __internal_functions__ limit 10;
  function_name | argument_types | return_type | function_type | deterministic |
description
-----+-----+-----+-----+-----+
+-----+
row_number      |                | bigint      | window        | true          |
rank            |                | bigint      | window        | true          |
dense_rank      |                | bigint      | window        | true          |
percent_rank    |                | double      | window        | true          |
cume_dist       |                | double      | window        | true          |
ntile           | bigint         | bigint      | window        | true          |
first_value     | bigint         | bigint      | window        | true          |
first_value     | double         | double      | window        | true          |
first_value     | boolean        | boolean     | window        | true          |
first_value     | varchar        | varchar     | window        | true          |
(10 rows)
```

9.1.2 Metadata

Metadata Schema 很简单，只有一个 Catalogs 表提供查询，可查询出当前 Presto 集群可用的 Catalog 数据源，如以下代码所示。

```
presto:information_schema> use system.metadata;
presto:metadata> show tables;
Table
-----
catalogs
(1 row)
```

显示出当前 Presto 集群所有可用的 Catalog 信息：

```
presto:metadata> select * from catalogs;
 catalog_name | connector_id
-----+-----
hive          | hive
system        | system
sqlserver     | sqlserver
jmx           | jmx
kafka         | kafka
mysql         | mysql
(6 rows)
```

9.1.3 Runtime

Runtime Schema 提供了当前 Presto 集群的运行信息，包括 Active Nodes、查询信息以及查询对应的 Tasks 信息，如以下代码所示。

```
presto:default> use system.runtime;
presto:runtime> show tables;
```


Table

nodes
queries
tasks
(3 rows)

- **Nodes:** 记录了当前 Presto 集群所有的 Active Nodes，可显示出节点地址、版本号、是否为 Coordinator 节点等信息。
- **Queries:** 记录了当前 Presto 的查询记录，包括历史和正在运行的 SQL 查询记录，但是 Presto 只会保留一部分历史查询记录，默认保留 100 个查询记录，如果超过 100 个，则将移除距离当前时间最久的查询。
- **Tasks:** 记录了查询对应的 Tasks 明细信息，包括 Split 信息、Task 运行的节点信息、运行时间、Task 处理的字节数等。

以下为上述列举的表的使用示例。

- 查询出 Presto 集群所有的节点信息。

```
presto:runtime> select * from nodes;
```

node_id	http_uri
6503c32b-cd96-4eac-87c5-586bbe67450a	http://192.168.100.10:8001
presto-main:0.107-1-g0cf7e34-dirty	true
73567813-152a-4e9c-87aa-587f66fccd2d	http://192.168.100.11:8001
presto-main:0.107-1-g0cf7e34-dirty	false
468cc226-5522-4d5a-8e8c-b0dade0a478c	http://192.168.100.12:8001
presto-main:0.107-1-g0cf7e34-dirty	false
cd30eddb-8cc1-4423-bb31-25ffe88d5f4d	http://192.168.100.13:8001
presto-main:0.107-1-g0cf7e34-dirty	false
9f586fbe-51a8-4bfb-a4f5-2750048ac915	http://192.168.100.14:8001
presto-main:0.107-1-g0cf7e34-dirty	false

(5 rows)

- 查询出 Presto 集群的查询记录。

```
presto:runtime> select * from queries limit 10;
```

node_id	query_id	state	user
6503c32b-cd96-4eac-87c5-586bbe67450a	20150908_094633_00047_zgnuq	FINISHED	hadoop
presto-cli	desc system.metadata.catalogs		
1	11		
6503c32b-cd96-4eac-87c5-586bbe67450a	20150908_075027_00010_zgnuq	FINISHED	hadoop
presto-cli	SHOW FUNCTIONS		
1	3		
6503c32b-cd96-4eac-87c5-586bbe67450a	20150907_085629_00063_zgnuq	FINISHED	hadoop


```
hadoop | presto-cli | select * from tasks order by query_id
|          1 |          7 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150907_085339_00057_zgnuq | FINISHED |
hadoop | presto-cli | select * from queries
|          1 |          6 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150907_100138_00070_zgnuq | FINISHED |
hadoop | presto-cli | SELECT table_name FROM information_schema.tables WHERE table_schema
= 'employees' |          1 |          5 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150908_101515_00055_zgnuq | FINISHED |
hadoop | presto-cli | select * from tables
|          1 |          4 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150908_102107_00068_zgnuq | FINISHED |
hadoop | presto-cli | show tables
|          1 |          4 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150907_085545_00060_zgnuq | FINISHED |
hadoop | presto-cli | select * from tasks where query_id='20150907_081600_00036_zgnuq'
|          1 |          9 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150908_030803_00009_zgnuq | FINISHED |
hadoop | presto-cli | select count(*) from employees
|          2 |          5 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150908_080334_00019_zgnuq | FINISHED |
hadoop | presto-cli | select * from hive.information_schema.tables
|          1 |          4 |
(10 rows)
```

- 查询 Query ID 为 20150908_102047_00062_zgnuq 的 Tasks 信息。

```
presto:runtime> select * from tasks where query_id='20150908_102047_00062_zgnuq';
          node_id          |          task_id          |
stage_id |          query_id          | state | splits | queued_splits |
running_splits | completed_splits | split_scheduled_time_ms | spli
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+
308312b3-bb32-45e4-b78a-96dc6c0fe894 | 20150908_102047_00062_zgnuq.0.0 |
20150908_102047_00062_zgnuq.0 | 20150908_102047_00062_zgnuq | FINISHED | 1 |
0 | 0 | 1 | 3 |
6503c32b-cd96-4eac-87c5-586bbe67450a | 20150908_102047_00062_zgnuq.1.0 |
20150908_102047_00062_zgnuq.1 | 20150908_102047_00062_zgnuq | FINISHED | 1 |
0 | 0 | 1 | 2 |
(2 rows)
```

9.2 System Connector 实现

System Connector 提供的内置信息，可更方便、快捷地管理 Presto 集群，涉及的源码主要在以下两个 package 内。

```
Presto-main
|-----src/main/java
|-----com.facebook.presto.connector.informationSchema
|-----com.facebook.presto.connector.system
```

本节将从源码层面剖析 System Connector 的实现细节。

9.2.1 Information_schema 实现

information_schema 包含了每个 Catalog 的基础元数据信息，例如可以查询到当前 Catalog 下所有的表信息及指定表的列信息，这些信息来源于当前 Catalog 初始化 Connector 时向 information_schema 注册了相关的管理类，详细在 com.facebook.presto.connector.ConnectorManager.addConnector 方法中：

```
private synchronized void addConnector(String catalogName, String connectorId,
Connector connector)
{
    .....
    // 当前 catalog 向 information_schema 注册 Metadata 信息 metadataManager.
    addInformationSchemaMetadata(makeInformationSchemaConnectorId(connectorId),
catalogName, new InformationSchemaMetadata(catalogName));
    // 当前 catalog 向 information_schema 注册 split 管理类
    splitManager.addConnectorSplitManager(makeInformationSchemaConnectorId(connectorId
), new InformationSchemaSplitManager(nodeManager));
    // 当前 catalog 向 information_schema 注册数据读取类，其中 metadataManager 是当前
catalog 的 metadata 管理类
    pageSourceManager.addConnectorPageSourceProvider(makeInformationSchemaConnect
orId(connectorId), new InformationSchemaPageSourceProvider(metadataManager));
    .....
}
```

在 com.facebook.presto.connector.informationSchema.InformationSchemaPageSourceProvider 中创建一个 ConnectorPageSource 对象，在查询 information_schema 下相关表的时候使用该对象提供的数据读取方法进行数据读取，如以下代码所示：

```
public ConnectorPageSource createPageSource(ConnectorSplit split,
List<ColumnHandle> columns)
{
    // 根据 Split 的不同返回具体的数据操作集合
    InternalTable table = getInternalTable(split, columns);
    // 根据 Columns 信息创建 page channels
    List<Integer> channels = new ArrayList<>();
    for (ColumnHandle column : columns) {
        String columnName = checkType(column, InformationSchemaColumnHandle.
class, "column").getColumnName();
        int columnIndex = table.getColumnIndex(columnName);
        channels.add(columnIndex);
    }
    // build pages 信息，page 的结构大致为 channel 和 block 组成的二维结构，具体结构可
以参照图 1-1
    ImmutableList.Builder<Page> pages = ImmutableList.builder();
    for (Page page : table.getPages()) {
        Block[] blocks = new Block[channels.size()];
        for (int index = 0; index < blocks.length; index++) {
            blocks[index] = page.getBlock(channels.get(index));
        }
    }
}
```

```

    }
    pages.add(new Page(page.getPositionCount(), blocks));
  }
  return new FixedPageSource(pages.build());
}

```

在上述 `getInternalTable` 方法中会依据 `Split` 所描述的 `Table` 信息（也就是实际要查询 `information_schema` 下的具体表），创建各自的数据结构。

```

private InternalTable getInternalTable(ConnectorSplit connectorSplit,
List<ColumnHandle> columns)
{
    InformationSchemaSplit split = checkType(connectorSplit,
InformationSchemaSplit.class, "split");

    checkNotNull(columns, "columns is null");

    InformationSchemaTableHandle handle = split.getTableHandle();
    Map<String, SerializableNativeValue> filters = split.getFilters();

    return getInformationSchemaTable(handle.getSession(),
handle.getCatalogName(), handle.getSchemaTableName(), filters);
}

public InternalTable getInformationSchemaTable(Session session, String
catalog, SchemaTableName table, Map<String, SerializableNativeValue> filters)
{
    if (table.equals(TABLE_COLUMNS)) {
        return buildColumns(session, catalog, filters);
    }
    if (table.equals(TABLE_TABLES)) {
        return buildTables(session, catalog, filters);
    }
    if (table.equals(TABLE_VIEWS)) {
        return buildViews(session, catalog, filters);
    }
    if (table.equals(TABLE_SCHEMATA)) {
        return buildSchemata(session, catalog);
    }
    if (table.equals(TABLE_INTERNAL_FUNCTIONS)) {
        return buildFunctions();
    }
    if (table.equals(TABLE_INTERNAL_PARTITIONS)) {
        return buildPartitions(session, catalog, filters);
    }

    throw new IllegalArgumentException(format("table does not exist: %s",
table));
}

```

在 `getInformationSchemaTable` 方法中所涉及的各个 `build` 方法，本质上调用的是

InformationSchemaPageSourceProvider 内定义的 Metadata metadata 所对应的相关方法，metadata 是 com.facebook.presto.connector.ConnectorManager.addConnector 方法中向 information_schema 注册的 metadata 管理类，也就是 Catalog 的 metadata 管理类：com.facebook.presto.metadata.MetadataManager，在 MetadataManager 中会根据 Catalog 映射到 connector 各自定义的 metadata 管理类中。例如要列出 Hive Catalog 中所有表：

```
select * from hive.information_schema.tables;
```

本质上还是会调用 com.facebook.presto.hive.HiveMetadata 中的 listTables 方法。

information_schema 包含的 Catalog 所有元数据信息均是以表的形式呈现的，例如 Columns、Tables 等信息也有存在初始化表的过程，只是这些表都是虚拟表，是在内存中构建的，这些表的初始化操作在 com.facebook.presto.connector.informationSchema.InformationSchemaMetadata 中完成，具体逻辑如以下代码所示：

```
// information_schema 所包含的表集合
public static final Map<SchemaTableName, ConnectorTableMetadata> TABLES =
schemaMetadataBuilder()
    // columns 表初始化
    .table(tableMetadataBuilder(TABLE_COLUMNS)
        .column("table_catalog", VARCHAR)
        .column("table_schema", VARCHAR)
        .column("table_name", VARCHAR)
        .column("column_name", VARCHAR)
        .column("ordinal_position", BIGINT)
        .column("column_default", VARCHAR)
        .column("is_nullable", VARCHAR)
        .column("data_type", VARCHAR)
        .column("is_partition_key", VARCHAR)
        .column("comment", VARCHAR)
        .build())
    // tables 表初始化
    .table(tableMetadataBuilder(TABLE_TABLES)
        .column("table_catalog", VARCHAR)
        .column("table_schema", VARCHAR)
        .column("table_name", VARCHAR)
        .column("table_type", VARCHAR)
        .build())
    // views 表初始化
    .table(tableMetadataBuilder(TABLE_VIEWS)
        .column("table_catalog", VARCHAR)
        .column("table_schema", VARCHAR)
        .column("table_name", VARCHAR)
        .column("view_definition", VARCHAR)
        .build())
    // schemata 表初始化
    .table(tableMetadataBuilder(TABLE_SCHEMATA)
        .column("catalog_name", VARCHAR)
        .column("schema_name", VARCHAR)
        .build())
    // __internal_functions__ 表初始化
```



```

        .table(tableMetadataBuilder(TABLE_INTERNAL_FUNCTIONS)
            .column("function_name", VARCHAR)
            .column("argument_types", VARCHAR)
            .column("return_type", VARCHAR)
            .column("function_type", VARCHAR)
            .column("deterministic", BOOLEAN)
            .column("description", VARCHAR)
            .build())
        // __internal_partitions 表初始化
        .table(tableMetadataBuilder(TABLE_INTERNAL_PARTITIONS)
            .column("table_catalog", VARCHAR)
            .column("table_schema", VARCHAR)
            .column("table_name", VARCHAR)
            .column("partition_number", BIGINT)
            .column("partition_key", VARCHAR)
            .column("partition_value", VARCHAR)
            .build())
    .build();

```

9.2.2 System Connector 实现

在 System Catalog 中, 除 information_schema schema 外, 还有 Metadata 和 Runtime 两个 Schema。System Catalog 的加载与其他的 Catalog 有所不同, 类似于 Hive, Mysql Catalog 的加载需要有 Catalog Properties 配置文件, 但是 System Connector 不需要, 在 com.facebook.presto.connector.system.SystemTablesModule 定义了 SystemTablesRegistrar 内部类, 直接调用了 com.facebook.presto.connector.ConnectorManager.createConnection 来创建 System Connector, 具体逻辑如以下代码所示:

```

    public void configure(Binder binder)
    {
        // 绑定 SystemTable 的实现类集合, 一共有 4 个, 分别对应的 metadata 和 runtime 两个 Schema
        // 下的 4 个表
        Multibinder<SystemTable> globalTableBinder =
            Multibinder.newSetBinder(binder, SystemTable.class);
        // 对应的是 runtime.nodes 表的操作类
        globalTableBinder.addBinding().to(NodeSystemTable.class).in(Scopes.SINGLETON);
        // 对应的是 runtime.queries 表的操作类
        globalTableBinder.addBinding().to(QuerySystemTable.class).in(Scopes.SINGLETON);
        // 对应的是 runtime.tasks 表的操作类
        globalTableBinder.addBinding().to(TaskSystemTable.class).in(Scopes.SINGLETON);
        // 对应的是 metadata.catalogs 表的操作类
        globalTableBinder.addBinding().to(CatalogSystemTable.class).in(Scopes.SINGLETON);

        binder.bind(SystemConnector.class).in(Scopes.SINGLETON);
        binder.bind(SystemTablesRegistrar.class).asEagerSingleton();
    }
    // System connector 注册器
    private static class SystemTablesRegistrar
    {
        @Inject

```

```

    public SystemTablesRegistrar(ConnectorManager manager, SystemConnector
connector)
    {
        // 创建 System connector
        manager.createConnection(SystemConnector.NAME, connector);
    }
}

```

System Connector 各个操作与其他类型的 Connector 操作无异，也是使用 Connector 定义的接口 ConnectorMetadata 的实现类 com.facebook.presto.connector.system.SystemTablesMetadata 来响应，最后也需要映射到 com.facebook.presto.spi.SystemTable 的各个实现类上。从上述源码中可以看到 SystemTable 有 4 个实现类：

com.facebook.presto.spi.SystemTable

```

|----com.facebook.presto.connector.system.CatalogSystemTable
|----com.facebook.presto.connector.system.NodeSystemTable
|----com.facebook.presto.connector.system.QuerySystemTable
|----com.facebook.presto.connector.system.TaskSystemTable

```

Metadata 和 runtime 两个 schema 的定义及其下属的表的定义均在这 4 个实现类进行了初始化的加载，其汇总入口在 SystemTablesMetadata 的构造方法中：

```

public SystemTablesMetadata(Set<SystemTable> tables)
{
    this.tables = tables.stream()
        .map(SystemTable::getTableMetadata)
        .collect(toMap(ConnectorTableMetadata::getTable, identity()));
}

```

其中 tables 即为 SystemTable 的实现类集合，每个实现类都定义了各自的 schema 及 table 信息，请看以下源码：

表 system.metadata.catalogs 的初始化在类 CatalogSystemTable 的静态属性初始化时完成，代码如下：

```

public static final SchemaTableName CATALOG_TABLE_NAME = new
SchemaTableName("metadata", "catalogs");

public static final ConnectorTableMetadata CATALOG_TABLE =
tableMetadataBuilder(CATALOG_TABLE_NAME)
    .column("catalog_name", VARCHAR)
    .column("connector_id", VARCHAR)
    .build();

```

表 system.runtime.nodes 的初始化在类 NodeSystemTable 的静态属性初始化时完成，代码如下：

```

public static final SchemaTableName NODES_TABLE_NAME = new
SchemaTableName("runtime", "nodes");

public static final ConnectorTableMetadata NODES_TABLE =

```

```

tableMetadataBuilder(NODES_TABLE_NAME)
    .column("node_id", VARCHAR)
    .column("http_uri", VARCHAR)
    .column("node_version", VARCHAR)
    .column("coordinator", BOOLEAN)
    .column("active", BOOLEAN)
    .build();

```

表 `system.runtime.queries` 的初始化在类 `QuerySystemTable` 的静态属性初始化时完成，代码如下：

```

public static final SchemaTableName QUERY_TABLE_NAME = new
SchemaTableName("runtime", "queries");

public static final ConnectorTableMetadata QUERY_TABLE =
tableMetadataBuilder(QUERY_TABLE_NAME)
    .column("node_id", VARCHAR)
    .column("query_id", VARCHAR)
    .column("state", VARCHAR)
    .column("user", VARCHAR)
    .column("source", VARCHAR)
    .column("query", VARCHAR)

    .column("queued_time_ms", BIGINT)
    .column("analysis_time_ms", BIGINT)
    .column("distributed_planning_time_ms", BIGINT)

    .column("created", TIMESTAMP)
    .column("started", TIMESTAMP)
    .column("last_heartbeat", TIMESTAMP)
    .column("end", TIMESTAMP)
    .build();

```

表 `system.runtime.tasks` 的初始化在类 `TaskSystemTable` 的静态属性初始化时完成，代码如下：

```

public static final SchemaTableName TASK_TABLE_NAME = new
SchemaTableName("runtime", "tasks");

public static final ConnectorTableMetadata TASK_TABLE =
tableMetadataBuilder(TASK_TABLE_NAME)
    .column("node_id", VARCHAR)

    .column("task_id", VARCHAR)
    .column("stage_id", VARCHAR)
    .column("query_id", VARCHAR)
    .column("state", VARCHAR)

    .column("splits", BIGINT)
    .column("queued_splits", BIGINT)
    .column("running_splits", BIGINT)
    .column("completed_splits", BIGINT)

```



```

        .column("split_scheduled_time_ms", BIGINT)
        .column("split_cpu_time_ms", BIGINT)
        .column("split_user_time_ms", BIGINT)
        .column("split_blocked_time_ms", BIGINT)

        .column("raw_input_bytes", BIGINT)
        .column("raw_input_rows", BIGINT)

        .column("processed_input_bytes", BIGINT)
        .column("processed_input_rows", BIGINT)

        .column("output_bytes", BIGINT)
        .column("output_rows", BIGINT)

        .column("created", TIMESTAMP)
        .column("start", TIMESTAMP)
        .column("last_heartbeat", TIMESTAMP)
        .column("end", TIMESTAMP)
        .build();

```

SystemTable 本身是 `com.facebook.presto.spi.RecordSet` 接口的子类，因此在 SystemTable 各个实现类中各自提供了对应的数据读取 `RecordCursor` 类，例如要查询 `system.metadata.catalogs` 表, `CatalogSystemTable.cursor()` 方法返回了用于遍历 `catalogs` 表中数据 `RecordCursor` 类，代码如下：

```

public RecordCursor cursor()
{
    // 内存表构造器
    Builder table = InMemoryRecordSet.builder(CATALOG_TABLE);
    for (Map.Entry<String, String> entry :
metadata.getCatalogNames().entrySet()) {
        // 遍历集群 catalog, 向内存表构造器增加表记录, entry.getKey 为 catalog //
name, entry.getValue 为 connectorId
        table.addRow(entry.getKey(), entry.getValue());
    }
    return table.build().cursor();
}

```

Nodes、Queries、Tasks 表的数据构建与上述代码描述的过程大同小异，区别在于各自的数据来源不同而已，例如：Nodes 数据来自于 `com.facebook.presto.metadata.InternalNodeManager` 接口实现类 `com.facebook.presto.metadata.DiscoveryNodeManager`, `NodeSystemTable.cursor()` 方法返回用于遍历 Nodes 表中数据的 `RecordCursor` 类，代码如下：

```

public RecordCursor cursor()
{
    Builder table = InMemoryRecordSet.builder(NODES_TABLE);
    AllNodes allNodes = nodeManager.getAllNodes();
    // 获取健康的节点信息
    for (Node node : allNodes.getActiveNodes()) {
        table.addRow(node.getNodeIdentifier(), node.getHttpUri().toString(),
getNodeVersion(node), isCoordinator(node), Boolean.TRUE);
    }
}

```



```

        for (Node node : allNodes.getInactiveNodes()) {
            table.addRow(node.getNodeIdentifier(), node.getHttpUri().toString(),
                getNodeVersion(node), isCoordinator(node), Boolean.FALSE);
        }
        return table.build().cursor();
    }
}

```

Queries 数据来自于接口：com.facebook.presto.execution.QueryManager 的实现类 com.facebook.presto.execution.SqlQueryManager，QuerySystemTable.cursor()方法返回用于遍历 Queries 表中数据的 RecordCursor 类，代码如下：

```

public RecordCursor cursor()
{
    Builder table = InMemoryRecordSet.builder(QUERY_TABLE);
    // 既包含了历史查询信息也包含了当前运行的查询信息
    for (QueryInfo queryInfo : queryManager.getAllQueryInfo()) {
        QueryStats queryStats = queryInfo.getQueryStats();
        table.addRow(
            nodeId,
            queryInfo.getQueryId().toString(),
            queryInfo.getState().toString(),
            queryInfo.getSession().getUser(),
            queryInfo.getSession().getSource(),
            queryInfo.getQuery(),

            toMillis(queryStats.getQueuedTime()),
            toMillis(queryStats.getAnalysisTime()),
            toMillis(queryStats.getDistributedPlanningTime()),

            toTimeStamp(queryStats.getCreateTime()),
            toTimeStamp(queryStats.getExecutionStartTime()),
            toTimeStamp(queryStats.getLastHeartbeat()),
            toTimeStamp(queryStats.getEndTime()));
    }
    return table.build().cursor();
}

```

Tasks 的数据则来自于接口 com.facebook.presto.execution.TaskManager 的实现类 com.facebook.presto.execution.SqlTaskManager，TaskSystemTable.cursor()方法返回用于遍历 Tasks 表中数据的 RecordCursor 类，代码如下：

```

public RecordCursor cursor()
{
    Builder table = InMemoryRecordSet.builder(TASK_TABLE);
    // 每个查询中包含的 task 详细信息
    for (TaskInfo taskInfo : taskManager.getAllTaskInfo()) {
        TaskStats stats = taskInfo.getStats();
        table.addRow(
            nodeId,

            taskInfo.getTaskId().toString(),
            taskInfo.getTaskId().getStageId().toString(),

```

```

        taskInfo.getTaskId().getQueryId().toString(),
        taskInfo.getState().toString(),

        (long) stats.getTotalDrivers(),
        (long) stats.getQueuedDrivers(),
        (long) stats.getRunningDrivers(),
        (long) stats.getCompletedDrivers(),

        toMillis(stats.getTotalScheduledTime()),
        toMillis(stats.getTotalCpuTime()),
        toMillis(stats.getTotalUserTime()),
        toMillis(stats.getTotalBlockedTime()),

        toBytes(stats.getRawInputDataSize()),
        stats.getRawInputPositions(),

        toBytes(stats.getProcessedInputDataSize()),
        stats.getProcessedInputPositions(),

        toBytes(stats.getOutputDataSize()),
        stats.getOutputPositions(),

        toTimeStamp(stats.getCreateTime()),
        toTimeStamp(stats.getFirstStartTime()),
        toTimeStamp(taskInfo.getLastHeartbeat()),
        toTimeStamp(stats.getEndTime());
    }
    return table.build().cursor();
}

```

Queries 和 Tasks 信息不会在内存中保存所有的历史数据，这一点在 9.1 节已经解释过，这里不再赘述。

9.3 小结

本章讲述了 System Connector 的实现过程。System Connector 记录了 Presto 集群的元数据信息，随着 Presto Server 的启动也随之启动。System Connector 针对每个 Catalog 提供了当前 Catalog 总体的表信息；同时也可以查询 Presto 集群的运行状况，例如节点信息、查询状态信息等，通过 System Connector 可以对集群的运行状况、资源使用情况，以及元数据信息进行准确、全面的了解和监控。

第 10 章

Hive Connector

在 Presto 提供的众多数据源 Connector 中，最常用的为 Hive Connector，并且也是查询性能最好的数据源之一。Hive Connector 使用 Hive 的元数据，Coordinator 节点通过 Hive Metastore 加载元数据信息，Presto 的计算节点读取 Hive 表对应的 HDFS 数据，整体架构如图 10-1 所示。

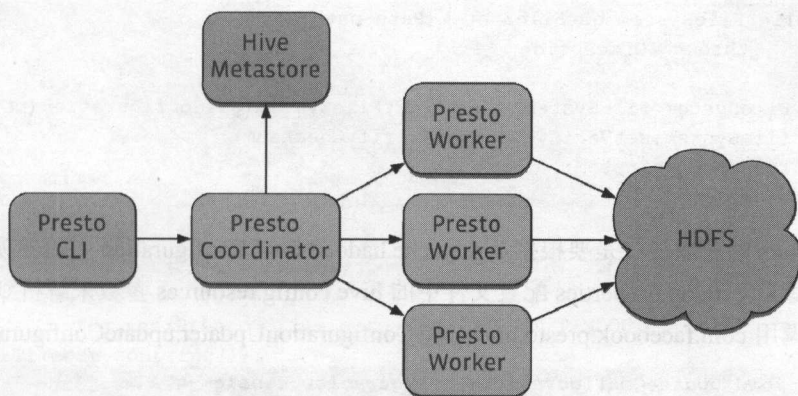


图 10-1

在 Presto 代码中，Hive Connector 占有比较大的比重，一共涉及以下几个模块：

```
Presto-root
|-----presto-orc
|-----presto-hive
|         |-----presto-hive-cdh4
|         |-----presto-hive-cdh5
|         |-----presto-hive-hadoop1
|         |-----presto-hive-hadoop2
```

其中 presto-orc 主要完成对 ORC 文件的各种操作。

10.1 与 Hive 的结合

Hive 可被视为一个数据仓库工具，其由三部分组成。

- 数据读取和存储：可将不同格式的数据文件存储在 HDFS 和 Amazon S3 中，也可以从中读取数据进行查询。
- 元数据管理：元数据保存的是数据文件和库表之间的映射关系、表的结构信息、表与库的关联关系等，元数据信息可保存在 MySQL 关系数据库或者是 Derby 嵌入式数据库中。
- 查询语句处理：Hive 提供了 HiveQL 查询语言，查询引擎分析查询语句后，将其转化成可执行的任务，最终生成 MapReduce 计算任务。

Hive Connector 只用到上述的前两个部分，Presto 本身提供了一套 SQL 查询引擎并且整个查询执行过程都在 Presto 框架内完成，除需要通过 Hive 的元数据服务 metaStore 获得表的元数据信息及与之关联的数据文件信息外，整个查询执行过程 Presto 与 Hive 没有任何其他交集。

在执行查询过程中，每个 Presto 计算节点相当于一个 Hadoop 客户端，需要从 HDFS 中拉取数据，在 Hive Connector 中均通过 `com.facebook.presto.hive.HdfsEnvironment` 操作 HDFS 的文件，如以下代码所示：

```
public FileSystem getFileSystem(Path path)
    throws IOException
{
    FileSystem fileSystem = path.getFileSystem(getConfiguration(path));
    fileSystem.setVerifyChecksum(verifyChecksum);
    return fileSystem;
}
```

FileSystem 的获取最终是要根据 `org.apache.hadoop.conf.Configuration` 中的各项配置来解析的，需要读取 `catalog properties` 配置文件中的 `hive.config.resources` 参数来解析到 HDFS 地址，最终会调用 `com.facebook.presto.hive.HdfsConfigurationUpdater.updateConfiguration` 方法：

```
public void updateConfiguration(Configuration config)
{
    // 将hive.config.resource 参数配置 hadoop 配置文件:core-site.xml 和 hdfs-site.xml
    加入到 configuration resource 中。
    if (resourcePaths != null) {
        for (String resourcePath : resourcePaths) {
            config.addResource(new Path(resourcePath));
        }
    }

    .....

    // 设置与 IPC 通信协议相关的超时时间、心跳间隔、失败重试等相关参数
    config.setInt("dfs.socket.timeout",
Ints.checkedCast(dfsTimeout.toMillis()));
    config.setInt("ipc.ping.interval",
```



```

Ints.checkedCast(dfsTimeout.toMillis()));
    config.setInt("ipc.client.connect.timeout",
Ints.checkedCast(dfsConnectTimeout.toMillis()));
    config.setInt("ipc.client.connect.max.retries", dfsConnectMaxRetries);

    // 支持 Amazon S3 文件系统的实现类
    config.set("fs.s3.impl", PrestoS3FileSystem.class.getName());
    config.set("fs.s3a.impl", PrestoS3FileSystem.class.getName());
    config.set("fs.s3n.impl", PrestoS3FileSystem.class.getName());
    config.set("fs.s3bfs.impl", "org.apache.hadoop.fs.s3.S3FileSystem");

    // Amazon S3 AccessKey 相关信息
    for (String scheme : ImmutableList.of("s3", "s3a", "s3bfs", "s3n")) {
        if (s3AwsAccessKey != null) {
            config.set(format("fs.%s.awsAccessKeyId", scheme),
s3AwsAccessKey);
        }
        if (s3AwsSecretKey != null) {
            config.set(format("fs.%s.awsSecretAccessKey", scheme),
s3AwsSecretKey);
        }
    }

    // Amazon S3 的相关配置
    config.setBoolean(PrestoS3FileSystem.S3_USE_INSTANCE_CREDENTIALS,
s3UseInstanceCredentials);
    config.setBoolean(PrestoS3FileSystem.S3_SSL_ENABLED, s3SslEnabled);
    config.setInt(PrestoS3FileSystem.S3_MAX_CLIENT_RETRIES,
s3MaxClientRetries);
    config.setInt(PrestoS3FileSystem.S3_MAX_ERROR_RETRIES,
s3MaxErrorRetries);
    config.set(PrestoS3FileSystem.S3_MAX_BACKOFF_TIME,
s3MaxBackoffTime.toString());
    config.set(PrestoS3FileSystem.S3_MAX_RETRY_TIME,
s3MaxRetryTime.toString());
    config.set(PrestoS3FileSystem.S3_CONNECT_TIMEOUT,
s3ConnectTimeout.toString());
    config.set(PrestoS3FileSystem.S3_SOCKET_TIMEOUT,
s3SocketTimeout.toString());
    config.set(PrestoS3FileSystem.S3_STAGING_DIRECTORY,
s3StagingDirectory.toString());
    config.setInt(PrestoS3FileSystem.S3_MAX_CONNECTIONS, s3MaxConnections);
    config.setLong(PrestoS3FileSystem.S3_MULTIPART_MIN_FILE_SIZE,
s3MultipartMinFileSize.toBytes());
    config.setLong(PrestoS3FileSystem.S3_MULTIPART_MIN_PART_SIZE,
s3MultipartMinPartSize.toBytes());

```

其中，`com.facebook.presto.hive.PrestoS3FileSystem` 是 Presto 用于支持 Amazon S3 文件系统的内置类，继承了 `org.apache.hadoop.fs.FileSystem`，内部实现了 `FileSystem` 操作文件的各种方法，只是换成了 Amazon S3 的相关 API。如果公司内部有类似的分布式存储系统作为 Presto 的数据源时，可以参考 Presto 读取 Amazon S3 的实现方式。

Presto 通过 Hive Metastore uri 来获取元数据, Hive Metastore uri 采用的是 Thrift 框架, Presto 作为客户端采用的是二进制的传输方式, Presto Thrift Client 实现在 `com.facebook.presto.hive.HiveMetastoreClient` 类中, 相关代码如下:

```
public HiveMetastoreClient(TTransport transport)
{
    super(new TBinaryProtocol(transport));
    this.transport = transport;
}
```

传输层 `org.apache.thrift.transport.TTransport` 在 `com.facebook.presto.hive.HiveMetastoreClientFactory.createTransport` 中创建:

```
protected TTransport createTransport(String host, int port)
    throws TTransportException
{
    TTransport transport;
    // 若没有 socket proxy 则直接用 host,port 进行连接
    if (socksProxy == null) {
        transport = new TTransportWrapper(new TSocket(host, port,
timeoutMillis), host);
        transport.open();
    }
    else {
        Socket socks = createSocksSocket(socksProxy);
        try {
            // 使用 proxy 来建立 socket 连接
            socks.connect(InetSocketAddress.createUnresolved(host, port),
timeoutMillis);
            socks.setSoTimeout(timeoutMillis);
        }
        catch (IOException e) {
            throw rewriteException(new TTransportException(e), host);
        }
        try {
            transport = new TTransportWrapper(new TSocket(socks), host);
        }
        catch (TTransportException e) {
            throw rewriteException(e, host);
        }
    }
    return transport;
}
```

Presto 对 Hive 元数据的操作均定义在 `com.facebook.presto.hive.metastore.HiveMetastore` 接口中, 定义的具体方法有:

```
// 创建表
void createTable(Table table);
// 删除表
void dropTable(String databaseName, String tableName);
// 重命名表
```

```

    void renameTable(String databaseName, String tableName, String newDatabaseName,
String newTableName);
    // 获取所有的 database
    List<String> getAllDatabases();
    // 获取指定 database 下的所有表
    List<String> getAllTables(String databaseName)
        throws NoSuchObjectException;
    // 获取指定 database 下的所有视图
    List<String> getAllViews(String databaseName)
        throws NoSuchObjectException;
    // 获取 database 实例
    Database getDatabase(String databaseName)
        throws NoSuchObjectException;
    // 获取指定表的分区名称
    List<String> getPartitionNames(String databaseName, String tableName)
        throws NoSuchObjectException;
    // 获取指定表实例
    Table getTable(String databaseName, String tableName)
        throws NoSuchObjectException;
    // 创建表分区
    Partition createPartition(String dbName, String tableName, List<String> values,
List<String> pCols, Table table, String location);
    // 向指定表增加分区
    int addPartitions(List<Partition> partitions, String dbName, String tblName);

```

HiveMetastore 接口的实现在 `com.facebook.presto.hive.HiveClientModule.configure` 方法中进行了实现的绑定：

```

    if (metastore != null) {
        binder.bind(HiveMetastore.class).toInstance(metastore);
    }
    else {
        binder.bind(HiveMetastore.class).to(CachingHiveMetastore.class).
in(Scopes.SINGLETON);
        newExporter(binder).export(HiveMetastore.class)
            .as(generatedNameOf(CachingHiveMetastore.class,
connectorId));
    }

```

其中，`metastore` 变量正常加载为空，不为空的情况是在测试用例中，因此，Hive Connector 元数据的加载都是在 `CachingHiveMetastore` 中实现的。在该类中使用了 Guava Cache 完成 Hive 元数据的加载和缓存，可设置过期和刷新时间间隔，具体参数如下。

- `hive.metastore-cache-ttl`：元数据缓存的过期时长，其含义为元数据从缓存创建算起，如果超过该参数设置的时长，则会强制刷新缓存值，默认时长为 1 小时。
- `hive.metastore-refresh-interval`：元数据缓存的刷新时间，其含义为访问元数据时距离上一次更新缓存的时长超过该参数，则将会更新缓存，默认 1 秒钟。

`CachingHiveMetastore` 中设置的缓存数据有：

```

// database name 数据
private final LoadingCache<String, List<String>> databaseNamesCache;

```



```
// Database 实例数据
private final LoadingCache<String, Database> databaseCache;
// 表名数据
private final LoadingCache<String, List<String>> tableNamesCache;
// 视图名数据
private final LoadingCache<String, List<String>> viewNamesCache;
// 分区名数据
private final LoadingCache<HiveTableName, List<String>> partitionNamesCache;
// 表实例数据
private final LoadingCache<HiveTableName, Table> tableCache;
// 分区实例数据
private final LoadingCache<HivePartitionName, Partition> partitionCache;
```

例如加载 Database 实例的时候,在 CachingHiveMetastore 构造方法中定义 LoadingCache 的 key 和 value, 如下:

```
databaseCache = CacheBuilder.newBuilder()
    .expireAfterWrite(expireAfterWriteMillis, MILLISECONDS)
    .refreshAfterWrite(refreshMills, MILLISECONDS)
    .build(asyncReloading(new CacheLoader<String, Database>()
    {
        @Override
        public Database load(String databaseName)
            throws Exception
        {
            return loadDatabase(databaseName);
        }
    }, executor));
```

loadDatabase 方法是真正加载 database 的地方:

```
private Database loadDatabase(final String databaseName)
    throws Exception
{
    try {
        return retry()
            .stopOn(NoSuchObjectException.class)
            .stopOnIllegalExceptions()
            .run("getDatabase", stats.getGetDatabase().wrap() -> {
                try (HiveMetastoreClient client =
                    clientProvider.createMetastoreClient()) {
                    return client.get_database(databaseName);
                }
            })
        ;
    }
    catch (NoSuchObjectException e) {
        throw e;
    }
    catch (TException e) {
        throw new PrestoException(HIVE_METASTORE_ERROR, e);
    }
}
```


在获取 Database 时，只需要按照以下方式即可：

```
public Database getDatabase(String databaseName)
    throws NoSuchObjectException
{
    return get(databaseCache, databaseName, NoSuchObjectException.class);
}

private static <K, V, E extends Exception> V get(LoadingCache<K, V> cache, K
key, Class<E> exceptionClass)
    throws E
{
    try {
        return cache.get(key);
    }
    catch (ExecutionException | UncheckedExecutionException e) {
        Throwable t = e.getCause();
        Throwables.propagateIfInstanceOf(t, exceptionClass);
        throw Throwables.propagate(t);
    }
}
```

其中 get 方法的处理逻辑如下：

如果当前缓存中可以匹配到 key，则直接从 LoadingCache 对象中将结果返回；如果没有匹配到，则会调用 loadDatabase 方法获得最新的 database。

10.2 Split 分片管理

Presto 在执行查询之前，都需要通过 Connector 自身的 Split 管理器将原始数据分割成若干个 Split，Hive Connector 依据 HDFS 文件的大小将数据分片，Hive Split 包含了读取 HDFS 的相关信息，例如数据路径、读取起始位置、数据偏移量等。首先介绍与 Hive Connector Split 管理相关的配置参数：

- `hive.max-split-size` : Split 分片的数据大小。
- `hive.max-initial-splits` : 初始 Split 分片的个数。
- `hive.max-initial-split-size` : 初始 Split 分片的数据大小，如果当前查询请求分配的分片个数还未达到初始的 Split 分片个数（`hive.max-initial-splits`），那么 Split 分片的数据大小为该参数的值，默认值为 `hive.max-split-size` 的二分之一。
- `hive.metastore.partition-batch-size.max` : 一个分区一个批次可以读取的最大分片个数。

Hive Connector Split 启用了后台任务来进行 Split 的加载，其入口函数为 `com.facebook.presto.hive.HiveSplitManager.getPartitionSplits(ConnectorTableHandle tableHandle, List<ConnectorPartition> connectorPartitions)`，代码如下：

```
public ConnectorSplitSource getPartitionSplits(ConnectorTableHandle
tableHandle, List<ConnectorPartition> connectorPartitions)
{
```

```

.....
SchemaTableName tableName = partition.getTableName();
// hive table bucket 分桶信息
Optional<HiveBucketing.HiveBucket> bucket = partition.getBucket();
// sort partitions
partitions = Ordering.natural().onResultOf(ConnectorPartition::
getPartitionId).reverse().sortedCopy(partitions);
Table table;
Iterable<HivePartitionMetadata> hivePartitions;
try {
    table = metastore.getTable(tableName.getSchemaName(),
tableName.getTableName());
    // 分区信息
    hivePartitions = getPartitionMetadata(table, tableName, partitions);
}
catch (NoSuchObjectException e) {
    throw new TableNotFoundException(tableName);
}
// Hive Split 加载器
HiveSplitLoader hiveSplitLoader = new BackgroundHiveSplitLoader(
    connectorId,
    table,
    hivePartitions,
    bucket,
    maxSplitSize,
    hiveTableHandle.getSession(),
    hdfsEnvironment,
    namenodeStats,
    directoryLister,
    executor,
    maxPartitionBatchSize,
    maxInitialSplitSize,
    maxInitialSplits,
    forceLocalScheduling,
    recursiveDfsWalkerEnabled);
// Hive Split 提供类
HiveSplitSource splitSource = new HiveSplitSource(connectorId,
maxOutstandingSplits, hiveSplitLoader);
hiveSplitLoader.start(splitSource);

return splitSource;
}

```

Source Stage Split 为分批次调度，Hive Connector 使用了后台异步线程来生成和管理 Split，通过接口调用 `com.facebook.presto.hive.HiveSplitSource.getNextBatch` 方法来获取 Split 集合：

```

public CompletableFuture<List<ConnectorSplit>> getNextBatch(int maxSize)
{
    checkState(!closed, "Provider is already closed");
    // maxSize 由 hive.max-split-size 参数设置
    CompletableFuture<List<ConnectorSplit>> future = queue.getBatchAsync

```

```

(maxSize);

    // 在调度过程中如果发生异常, 为防止异常而阻塞整个查询, 因此需要抛出异常来终止查询
    if (throwable.get() != null) {
        return failedFuture(throwable.get());
    }

    // Split 个数小于预设的值, 执行 HiveSplitLoader.resume 方法继续加载 Split
    // maxOutstandingSplits 由 hive.max-outstanding-splits 参数设置
    future.thenAccept(splits -> {
        if (outstandingSplitCount.addAndGet(-splits.size()) <
maxOutstandingSplits) {
            // we are below the low water mark (and there isn't a failure) so
resume scanning hdfs
            splitLoader.resume();
        }
    });

    return future;
}

```

在分析 Split 加载机制之前, 首先需要了解 HiveSplitSource 定义的几个重要变量。

- `private final AsyncQueue<ConnectorSplit> queue` : 一个简单的无界限队列, 提供了可异步批量获取队列中元素的方法。
- `private final AtomicInteger outstandingSplitCount` : 全局的 Split 计数器, 记录了当前已经生成但未分配的 Split 个数。
- `private final int maxOutstandingSplits` : 允许加载的 Split 最大个数, 默认为 1000。
- `private final HiveSplitLoader splitLoader` : Hive Split 加载器。

其中 HiveSplitLoader Split 加载器承载了 Hive Split 整个生成过程, 其实现类是 `com.facebook.presto.hive.BackgroundHiveSplitLoader`, 加载 Split 入口即为 `start` 方法:

```

public void start(HiveSplitSource splitSource)
{
    this.hiveSplitSource = splitSource;
    startLoadSplits();
}

```

整个生成过程大致如下:

- 根据文件的存储格式判断该数据文件是否可分割 (`isSplittable`), 例如 LZO 压缩格式的文件如果没有生成索引文件, 则不可分割, 只能一个文件形成一个 Split。
- 获取 Hive Connector 定义的 split max size, 默认为 64 MB, 如果分片的个数还未达到初始的分片 `hive.max-initial-splits` 所定义的个数, 那么 split max size 等于 `hive.max-initial-split-size` 的值, 在默认情况下, `hive.max-initial-split-size` 等于 `hive.max-split-size` 参数值的二分之一。
- 计算数据文件切分成块的个数, 并计算数据块的大小。
- 从数据文件的起始位置开始, 计算每个块的结束位置, 结束位置为当前块的起始位

置加上数据块大小，那么当前块的结束位置也就是下一个块的起始位置，如此一直循环，直到数据文件的最终结束位置。

上述逻辑的执行代码为：`com.facebook.presto.hive.BackgroundHiveSplitLoader.createHiveSplits()`方法，Presto 会针对每个文件都调用一次该方法，该方法的内容如下：

```
private List<HiveSplit> createHiveSplits(
    String partitionName,
    String path,
    BlockLocation[] blockLocations,
    long start,
    long length,
    Properties schema,
    List<HivePartitionKey> partitionKeys,
    boolean splittable,
    ConnectorSession session,
    TupleDomain<HiveColumnHandle> effectivePredicate)
    throws IOException
{
    ImmutableList.Builder<HiveSplit> builder = ImmutableList.builder();

    // 是否允许远程读取数据块的标识
    boolean forceLocalScheduling =
HiveSessionProperties.getForceLocalScheduling(session, this.forceLocalScheduling);
    // 该数据文件是可分片
    if (splittable) {
        for (BlockLocation blockLocation : blockLocations) {
            // 获取 block 的所在节点的地址信息
            List<HostAddress> addresses = toHostAddress(blockLocation.getHosts());
            // 获取分片大小
            long maxBytes = maxSplitSize.toBytes();
            // 判断是否已经达到初始分片个数，若没有则分片大小取初始分片大小
            if (remainingInitialSplits.get() > 0) {
                maxBytes = maxInitialSplitSize.toBytes();
            }

            // 计算块的个数
            int chunks = Math.max(1, (int) (blockLocation.getLength() /
maxBytes));

            // 计算每个块的大小
            long targetChunkSize = (long) Math.ceil(blockLocation.getLength()
* 1.0 / chunks);
            // 对数据文件进行切分
            long chunkOffset = 0;
            while (chunkOffset < blockLocation.getLength()) {
                long chunkLength = Math.min(targetChunkSize,
blockLocation.getLength() - chunkOffset);
                // 创建 Hive Split 信息
                builder.add(new HiveSplit(connectorId,
                    table.getDbName(),
                    table.getTableName(),
```



```

        partitionName,
        path,
        blockLocation.getOffset() + chunkOffset,
        chunkLength,
        schema,
        partitionKeys,
        addresses,
        forceLocalScheduling,
        session,
        effectivePredicate));

        chunkOffset += chunkLength;
        remainingInitialSplits.decrementAndGet();
    }
    checkState(chunkOffset == blockLocation.getLength(), "Error
splitting blocks");
}
}
else {
    // 如果数据文件不能分片, 则一个文件一个分片
    List<HostAddress> addresses = ImmutableList.of();
    if (blockLocations.length > 0) {
        addresses = toHostAddress(blockLocations[0].getHosts());
    }

    builder.add(new HiveSplit(connectorId,
        table.getDbName(),
        table.getTableName(),
        partitionName,
        path,
        start,
        length,
        schema,
        partitionKeys,
        addresses,
        forceLocalScheduling,
        session,
        effectivePredicate));
}
return builder.build();
}
}

```

10.3 数据读取

HDFS 存储的数据文件格式多种多样, Presto-0.107 版本支持的数据格式有以下几种。

- ORC: 即 Optimized Row Columnar file, 列式存储引擎, 源自于 RCFile, 对 RCFile 进行了优化改进, 在新版本 Hive 中支持对 ORC 文件的 Update 操作, 其目前是 Apache 顶级项目, 其官方网站为 <http://orc.apache.org/>。
- DWRF: ORC File 的分支之一, 由 Facebook 主导, Github 地址为: <https://github.com/>

facebook/hive-dwrf。

- PARQUET: 其在设计之初用于存储嵌套式数据, 例如 Protobuf、Thrift、Json 等, 将这类数据以列式格式存储, 使用 PARQUET 操作这类数据有着更加高效的 IO 效率, 目前 PARQUET 也是 Apache 顶级项目, 其官方网站为 <http://parquet.apache.org/>。
- RCFile : 即 Record-Columnar File Format, 列式存储格式, 最初由 Facebook 主导开发, 并适用于 Hive 中。
- SEQUENCEFILE: 二进制格式存储的文件。
- TEXTFILE: 文本文件格式。

Presto 针对 ORC 做了很多的优化, 例如 Predicate Down 和 Lazy Load 等优化, 使得在 Presto 中查询 ORC 格式的数据时, 具有很好的查询效率。

Presto 在支持这么多存储格式的同时, 也支持 Gzip、Snnapy 等压缩, 但是默认不支持 LZO 压缩。

Hive Connector 在进行数据读取的时候, 根据数据文件的存储格式, 提供了两种实现方式: `com.facebook.presto.spi.RecordCursor` 和 `com.facebook.presto.spi.ConnectorPageSource`, 这两种实现方式均在类 `com.facebook.presto.hive.HiveClientModule` 中进行绑定初始化, 如下代码所示:

```
// HiveRecordCursorProvider 定义了 createHiveRecordCursor 方法
Multibinder<HiveRecordCursorProvider> recordCursorProviderBinder = Multibinder.
newSetBinder(binder, HiveRecordCursorProvider.class);
recordCursorProviderBinder.addBinding().to(OrcRecordCursorProvider.class).in(
Scopes.SINGLETON);
recordCursorProviderBinder.addBinding().to(ParquetRecordCursorProvider.class)
.in(Scopes.SINGLETON);
recordCursorProviderBinder.addBinding().to(DwrfRecordCursorProvider.class).in(
Scopes.SINGLETON);
recordCursorProviderBinder.addBinding().to(ColumnarTextHiveRecordCursorProvid
er.class).in(Scopes.SINGLETON);
recordCursorProviderBinder.addBinding().to(ColumnarBinaryHiveRecordCursorProv
ider.class).in(Scopes.SINGLETON);
recordCursorProviderBinder.addBinding().to(GenericHiveRecordCursorProvider.cl
ass).in(Scopes.SINGLETON);
// HivePageSourceFactory 定义了 createPageSource 方法
Multibinder<HivePageSourceFactory> pageSourceFactoryBinder = Multibinder.
newSetBinder(binder, HivePageSourceFactory.class);
pageSourceFactoryBinder.addBinding().to(RcFilePageSourceFactory.class).in(Sco
pes.SINGLETON);
pageSourceFactoryBinder.addBinding().to(OrcPageSourceFactory.class).in(Scopes
.SINGLETON);
pageSourceFactoryBinder.addBinding().to(DwrfPageSourceFactory.class).in(Scope
s.SINGLETON);
```

其中, `HiveRecordCursorProvider` 接口定义的方法是返回一个 `RecordCursor` 的实现类, `RecordCursor` 的实现类在 `Hive Connector` 中也根据存储格式定义了不同的实现类, 关系如图 10-2 所示。

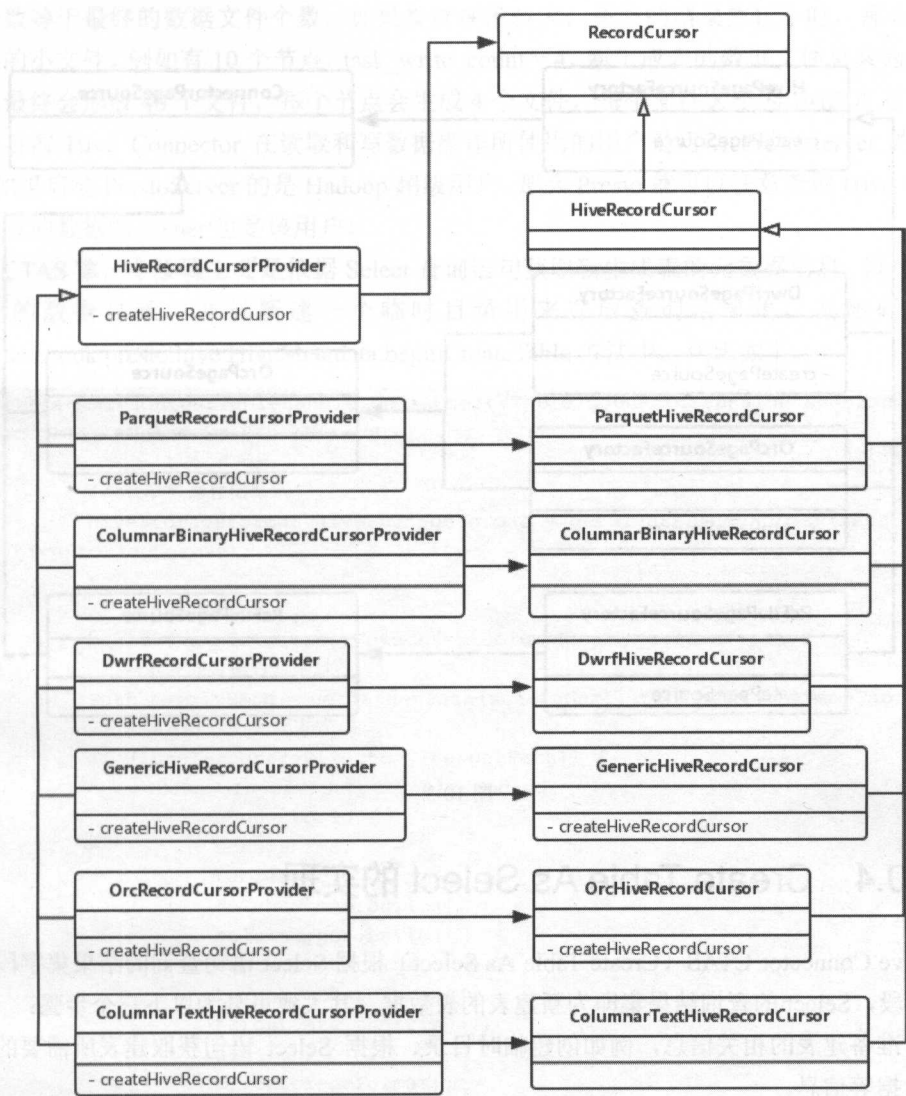


图 10-2

HivePageSourceFactory 接口也根据存储格式定义了若干个形成 Page Source 的工厂类，但是只针对了 ORC File 和 RCFile 两种格式，其余的格式未做处理，如图 10-3 所示。

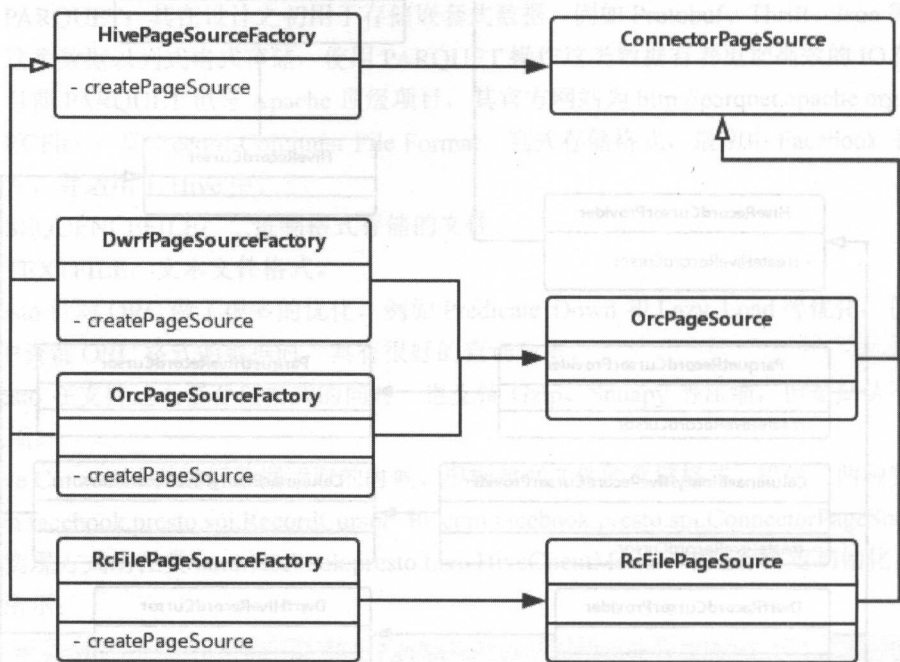


图 10-3

10.4 Create Table As Select 的实现

Hive Connector CTAS (Create Table As Select) 根据 Select 语句查询的结果集字段作为建表字段，Select 的查询结果集即为新建表的表数据，其大致可分为以下三个步骤：

- 准备建表的相关信息，例如创建临时目录、根据 Select 语句获取建表所需要的元数据等信息。
- 将 Select 查询语句的查询结果写入到指定临时目录中。
- 执行创建表操作，并将临时目录中的数据文件移动到表对应的目录下，若有分区信息，则同时也会创建分区信息，并更新元数据。

在以上的三个步骤中，第一步和第三步在 Coordinator 上执行，而第二步将查询结果集写入到指定的 HDFS 临时目录的过程每个计算节都将参与，由 `com.facebook.presto.operator.TableWriterOperator` 来驱动，写入 HDFS 数据的线程个数可由参数 `task.writer-count` 来设置，默认值为 1，也可以通过 session 参数 `task_writer_count` 来动态设置该参数，在命令行里面设置方式为：

```
set session task_writer_count = 2
```

需要注意的是，最终形成的数据文件个数也和该参数有关系，在计算节点上写入线程

的个数等于最终的数据文件个数。如果集群规模较大, 最终的结果集较小时, 那么会形成很多的小文件。例如有 10 个节点, `task_write_count=4`, 新生成表的数据文件总共为 40MB, 那么最终会形成 40 个文件, 每个节点会生成 4 个文件, 每个文件大小为 1MB 左右。

目前 Hive Connector 在读取和写数据操作所使用的用户是启动 PrestoServer 的系统用户, 如果启动 PrestoServer 的是 Hadoop 超级用户, 那么 Presto 就可以任意查询 Hive 中的表, 新生成的数据的 owner 也是该用户。

CTAS 第一个步骤主要是依据 Select 查询语句获取新生成表的元数据信息, 以及创建表对应的数据目录, 并且新建一个临时目录用来存放查询结果集, 其源码实现在 `com.facebook.presto.hive.HiveMetadata.beginCreateTable` 方法中, 代码如下:

```
public HiveOutputTableHandle beginCreateTable(ConnectorSession session,
ConnectorTableMetadata tableMetadata)
{
    // 获取数据存储格式
    HiveStorageFormat hiveStorageFormat = getHiveStorageFormat(session,
this.hiveStorageFormat);
    .....
    // 获取新建表的列信息
    buildColumnInfo(tableMetadata, columnNames, columnTypes);
    // 获取表的数据目录地址
    Path targetPath = getTargetPath(schemaName, tableName, schemaTableName);
    // Amazon S3 处理方式
    if (!useTemporaryDirectory(targetPath)) {
        return new HiveOutputTableHandle(
            connectorId,
            schemaName,
            tableName,
            columnNames.build(),
            columnTypes.build(),
            tableMetadata.getOwner(),
            targetPath.toString(),
            targetPath.toString(),
            session,
            hiveStorageFormat);
    }

    // 生成临时目录
    String temporaryPrefix = "/tmp/presto-" + StandardSystemProperty.
USER_NAME.value();
    Path temporaryRoot = new Path(targetPath, temporaryPrefix);
    Path temporaryPath = new Path(temporaryRoot, randomUUID().toString());
    createDirectories(temporaryPath);
    return new HiveOutputTableHandle(
        connectorId,
        schemaName,
        tableName,
        columnNames.build(),
        columnTypes.build(),
        tableMetadata.getOwner(),
```

```
targetPath.toString(),
temporaryPath.toString(),
session,
hiveStorageFormat];
}
```

查询结果集写入过程在 `com.facebook.presto.hive.HiveRecordSink` 中完成，其实现思路为：根据数据存储格式，创建数据写入器，将结果集写入到临时目录中：

```
public HiveRecordSink(HiveOutputTableHandle handle, Path target, JobConf conf)
{
    // 列的个数
    fieldCount = handle.getColumnNames().size();
    // 列的相关信息
    sampleWeightField = handle.getColumnNames().indexOf(SAMPLE_WEIGHT_
COLUMN_NAME);
    columnTypes = ImmutableList.copyOf(handle.getColumnTypes());
    connectorSession = handle.getConnectorSession();
    hasDateTimeTypes = handle.getColumnTypes().stream().map(this::
containsDateTime).collect(toList());
    // 转为 Hive 的列类型
    Iterable<String> hiveTypeNames = transform(transform(handle.
getColumnTypes(), HiveType::toHiveType), HiveType::getHiveTypeName);
    // 设置列的相关元数据参数
    Properties properties = new Properties();
    properties.setProperty(META_TABLE_COLUMNS,
Joiner.on(',').join(handle.getColumnNames()));
    properties.setProperty(META_TABLE_COLUMN_TYPES,
Joiner.on(':').join(hiveTypeNames));
    // 序列化
    serializer = initializeSerializer(conf, properties,
handle.getHiveStorageFormat().getSerDe());
    // 创建数据写入器
    recordWriter = createRecordWriter(target, conf, properties,
handle.getHiveStorageFormat().getOutputFormat());

    tableInspector =
getStandardStructObjectInspector(handle.getColumnNames(),
getJavaObjectInspectors(columnTypes));
    structFields =
ImmutableList.copyOf(tableInspector.getAllStructFieldRefs());
    row = tableInspector.create();
}
```

数据写入到临时目录完成后，将进入第三个环节——真正执行创建表的操作，并且将临时目录的数据移动到表对应的数据目录地址，然后创建目标表的元数据。源码实现在 `com.facebook.presto.hive.HiveMetadata.commitCreateTable` 方法中，代码如下：

```
public void commitCreateTable(ConnectorOutputTableHandle tableHandle,
Collection<Slice> fragments)
{
    .....
```

```

// 将临时目录的数据移动到表对应的数据目录地址
if (handle.hasTemporaryPath()) {
    if (pathExists(targetPath)) {
        SchemaTableName table = new SchemaTableName(handle.getSchemaName(),
handle.getTableName());
        throw new PrestoException(HIVE_PATH_ALREADY_EXISTS, format("Unable
to commit creation of table '%s': target directory already exists: %s", table,
targetPath));
    }
    rename(new Path(handle.getTemporaryPath()), targetPath);
}

// 获取列名
List<String> types = handle.getColumnTypes().stream()
    .map(HiveType::toHiveType)
    .map(HiveType::getHiveTypeName)
    .collect(toList());

// 构建表的列信息
boolean sampled = false;
ImmutableList.Builder<FieldSchema> columns = ImmutableList.builder();
for (int i = 0; i < handle.getColumnNames().size(); i++) {
    String name = handle.getColumnNames().get(i);
    String type = types.get(i);
    if (name.equals(SAMPLE_WEIGHT_COLUMN_NAME)) {
        columns.add(new FieldSchema(name, type, "Presto sample weight
column"));
        sampled = true;
    }
    else {
        columns.add(new FieldSchema(name, type, null));
    }
}

// 数据存储格式
HiveStorageFormat hiveStorageFormat = handle.getHiveStorageFormat();
// 以下操作都是为了在 Hive 的元数据中创建表拼装了相关信息
// 数据序列化
SerDeInfo serdeInfo = new SerDeInfo();
serdeInfo.setName(handle.getTableName());
serdeInfo.setSerializationLib(hiveStorageFormat.getSerDe());
serdeInfo.setParameters(ImmutableMap.<String, String>of());
// 定义 inputFormat 和 outputFormat
StorageDescriptor sd = new StorageDescriptor();
sd.setLocation(targetPath.toString());
sd.setCols(columns.build());
sd.setSerdeInfo(serdeInfo);
sd.setInputFormat(hiveStorageFormat.getInputFormat());
sd.setOutputFormat(hiveStorageFormat.getOutputFormat());
sd.setParameters(ImmutableMap.<String, String>of());
// 定义表名、表的列信息、表的所属用户等
Table table = new Table();
table.setDbName(handle.getSchemaName());
table.setTableName(handle.getTableName());

```



```

        table.setOwner(handle.getTableOwner());
        table.setTableType(TableType.MANAGED_TABLE.toString());
        String tableComment = "Created by Presto";
        if (sampled) {
            tableComment = "Sampled table created by Presto. Only query this table
from Hive if you understand how Presto implements sampling.";
        }
        table.setParameters(ImmutableMap.of("comment", tableComment));
        table.setPartitionKeys(ImmutableList.<FieldSchema>of());
        table.setSd(sd);
        // 调用 hive metastore 接口创建表
        metastore.createTable(table);
    }

```

10.5 小结

本章从 Presto 与 Hive 的结合入手，详细讲述了 Presto 通过 Hive MetaStore 服务使用和管理 Hive 中表的元数据，并且对 Hive 的分片管理以及数据读取过程从源码层面进行了解读。Hive Connector 有很大一部分代码用于针对各种数据格式进行数据读取优化，例如 ORC、PARQUET 等格式的读取优化，因此本章还对 Hive 中的数据读取部分进行了详细讲解。最后描述了 Presto 中 CTAS 的实现原理，并进行了代码解读。通过本章，读者可以深刻理解 Presto 的 Hive Connector 的内部架构和实现原理。

第 11 章

Kafka Connector

11.1 认识 Kafka Connector

11.1.1 配置

Kafka Connector 支持 Apache Kafka (<http://kafka.apache.org/>) 0.8 及以上版本, 强烈推荐
使用 0.8.1 或者更高的版本。

Presto 通过 Kafka Connector 将 Apache Kafka 中的 topics 当作表进行处理, topics 中的每
条消息在 Presto 中被解析为表中的一行数据。

topics 可以是实时的, 当 topics 中的数据加载到 Presto 中时, 对应的 rows 就可以被查
询到, 当 topics 中的 segments 被删除后, 对应的 rows 就不会出现在查询结果中。这样的话,
如果在一次查询中 (例如执行一个自连接操作) 多次访问同一张表, 就可能会产生奇怪的
查询结果。

在 Presto 中配置 Kafka Connector, 需要在 Presto 安装目录中创建一个 Catalog 属性文件:
etc/catalog/kafka.properties, 该文件包含以下内容, 根据实际情况将属性值替换为合适的值:

```
connector.name=kafka
kafka.table-names=table1,table2
kafka.nodes=host1:port,host2:port
```

你可以根据需要配置多个 Catalogs, 如果你有其他的 Kafka 集群, 简单地添加另一个不
同名字的属性文件到 etc/catalog 目录下即可 (确保属性文件以 .properties 结尾)。例如, 如果
你将属性文件命名为 sales.properties, 则 Presto 将会根据配置的 Connector 信息创建一个名字
为 sales 的 Catalog。

11.1.2 配置属性

表 11-1 所示的配置属性是可用的。

表 11-1

属性名称	描 述
kafka.table-names	列出 Catalog 提供的所有表的表名
kafka.default-schema	表的默认 Schema
kafka.nodes	Kafka 集群的节点列表
kafka.connect-timeout	连接到 Kafka 集群的超时时间
kafka.buffer-size	Kafka 读入缓冲区的大小
kafka.table-description-dir	包含 topic 描述文件的目录位置
kafka.hide-internal-columns	是否隐藏表的内部列

下面对上述几个配置选项做进一步描述。

1. kafka.table-names

kafka.table-names 属性值为当前 Catalog 提供的所有表的列表，表名之间以逗号分隔。一个表名可以不限定 schema 的名字（简单表名），这样的表会被加入到默认的 schema 中，也可以限定 schema 的名字（<schema-name>.<table-name>）。

这里定义的每一个表都可以定义一个对应的表描述文件。如果没有表描述文件，表的名字将用来作为 Kafka 集群中 topic 的名字，这样的话在 topic 中没有数据列会被映射到表中。这个表仍会包含所有的内置列。

该属性是必须配置的，没有默认值，必须至少设定一个表名。

2. kafka.default-schema

该属性定义 schema 的名字，该 schema 包含 kafka.table-names 属性中定义的所有没有限定 schema 名字的 table。

该属性是可选的，默认值是 default。

3. kafka.nodes

该属性值是以逗号分隔的一系列 hostname:port 对的列表。

该属性是必须配置的，没有默认值并且至少设定一个 node 节点。

注意：即使在该属性的配置中只配置了 Kafka 集群的一个子集，也必须要保证 Presto 能够连接到 Kafka 集群中所有的节点，这是因为 Kafka 中的 segment 文件可能位于某个特定的 Kafka 节点之上，要查询到这部分数据，就要保证 Presto 可以连接到该节点。

4. kafka.connect-timeout

该属性为连接到 Kafka 节点的超时时间。一个正在执行操作的 Kafka 集群在接受新的连接之前可能需要比较长的时间；当出现因为超时而造成查询失败时，选择增大该属性的值是一个好的办法。

该属性是可选的，默认值是 10 秒。

注意：设置该属性的值时，格式是数值+时间单位，例如 10.5s，时间单位可以是 d（天）、h（小时）、m（分钟）、s（秒）、ms（毫秒）、us（微秒）、ns（纳秒）。

5. kafka.buffer-size

该属性是指 Presto 为了从 Kafka 中读取数据而设定的内部数据缓冲区的大小。数据缓冲区必须能够容纳至少一条消息，在理想情况下可以容纳多条消息。每一个 worker 节点都会分配一个数据缓冲区。

该属性是可选的，默认值为 64KB。

注意：设置该属性的值时，其格式为数值+存储单位，例如 64.5KB，存储单位可以是 B（字节）、KB（千字节）、MB（兆字节）、GB（吉字节）、TB（太字节）、PB（拍字节）。

6. kafka.table-description-dir

该属性设置一个文件夹来存放一个或多个 JSON 文件（文件必须以.json 结尾），这些文件的内容是表的描述信息。

该属性是可选的，默认是 etc/kafka。

7. kafka.hide-internal-columns

除在表的描述性文件里面定义的数据列外，Kafka Connector 还为每个表维护了一些额外的列。如果这些额外的列被隐藏，则在查询的时候仍然可以对这些隐藏的列进行查询，但是在使用 DESCRIBE <table-name> 或者 SELECT * 进行查询时，这些内部列不会显示。

11.1.3 内置字段

对于属性文件中定义的每一个表，在默认情况下，Kafka 连接器都会为该表创建如表 11-2 所示的内置字段。

表 11-2

列 名	类 型	描 述
_partition_id	BIGINT	在 Kafka 中包含当前行的 partition 的 ID
_partition_offset	BIGINT	当前行在 kafka partition 中的偏移量（该行 message 起始位置在所在的 partition 中的偏移量，取值为 0~（该 partition 中 message 总数-1））
_segment_start	BIGINT	该行所在的 segment 的最小偏移量，该偏移量具体到每个 partition。该值是该行所在的 segment 的第一条 message 在 partition 中的偏移量，该行 message 所在的 segment 的文件名就是第一条 message 的偏移量。 在每个 partition 目录下面，message 是分段存储的，包括一个数据文件以.log 结尾，和一个索引文件以 .index 结尾，文件名为该 segment 中第一条 message 在 partition 中的偏移量
_segment_end	BIGINT	该行所在的 segment 的最大偏移量，该偏移量具体到每个 partition。该值与下一个 segment（如果存在）的 _segment_start 的值相同
_segment_count	BIGINT	当前行在 segment 中的数值，即当前行是所在 segment 的第几条 message，对于未压缩的 topic： $_segment_start + _segment_count = _partition_offset + 1$

续表

列 名	类 型	描 述
_message_corrupt	BOOLEAN	如果译码器不能解析该行信息，则该值为 true。当该值为 true 时，则从 message 中映射出来的数据行的信息应被认为是无效的
_message	VARCHAR	message 对应的 UTF-8 编码的字符串，该字符串只对内容为文本的 topic 有用
_message_length	BIGINT	该条 message 所占的字节数
_key_corrupt	BOOLEAN	如果 key 的译码器不能解析 key 值，则该值为 true。当该值为 true 时，key 列对应的信息应被认为是无效的
_key	VARCHAR	key 值对应的 UTF-8 编码的字符串，只有当 key 是文本时该值有用
_key_length	BIGINT	key 所占的字节数

如果表没有对应的表定义文件，则内部列 `_key_corrupt` 和 `_message_corrupt` 的值为 `false`。

11.1.4 表定义文件

在 Kafka 的 topics 中是以字节的形式来保存消息的，至于消息的定义和解析则是由生产者和消费者来决定的。对 Presto 来说，当对这些数据进行查询的时候，这些数据应该映射到不同的列中。

topics 中的数据是 JSON 格式的文本内容，如果 `_message` 字段的字节数据是要映射为 UTF-8 格式的字符串，则完全可以不使用任何表定义文件，而是使用 Presto 自带的 JSON 函数来对 `_message` 列进行解析。但是这样会非常麻烦，查询语句会非常冗长且很难编写。

一个表定义文件内容为对一个表的 JSON 格式的定义，如表 11-3 所示。表定义文件的名字可以是任意的，但必须以 `.json` 结尾。一个表定义文件的内容格式如下：

```
{
  "tableName": ...,
  "schemaName": ...,
  "topicName": ...,
  "key": {
    "dataFormat": ...,
    "fields": [
      ...
    ]
  },
  "message": {
    "dataFormat": ...,
    "fields": [
      ...
    ]
  }
}
```


表 11-3

字 段	是否必需	类型	描 述
tableName	必需	String	该文件描述的 topic 在 Presto 中对应的表名
schemaName	可选	String	Presto 中该表的 schema 名字。如果省略，则会使用默认的 schema 名字
topicName	必需	String	对应的 Kafka 的 topic 名字
key	可选	JSON	Message key 与表中各个字段的映射关系的定义
message	可选	JSON	Message 本身与表中各个字段的映射关系的定义

11.1.5 Kafka 中的 key 和 message

从 Kafka 0.8 开始，topic 中的每条 message 的 key 都是可选的。在表的定义文件中可以同时包含 key 和 message 的数据内容到表中列的映射关系的定义。

在表定义文件中，key 和 message 两个字段各自的定义部分也是一个 JSON 对象，并且对于这两个字段每个字段又必须包含如表 11-4 所示的两个字段。表定义文件中可以包含的字段及其含义如表 11-4 所示。

表 11-4

字 段	是否必需	类型	描 述
dataFormat	必需	String	为该组 fields 选择译码器
fields	必需	JSON	字段定义的列表，每一个字段定义会在 Presto 对应表中创建一个新的列

fields 字段是一个 JSON 数组，数组中的每一个字段又是一个 JSON 对象：

```
{
  "name": ...,
  "type": ...,
  "dataFormat": ...,
  "mapping": ...,
  "formatHint": ...,
  "hidden": ...,
  "comment": ...
}
```

JSON 数组中可以包含的字段及其含义如表 11-5 所示。

表 11-5

字段	是否必需	类型	描 述
name	必需	string	Presto 中表的列名
type	必需	string	Presto 中该列的类型
dataFormat	可选	string	为该列选择译码器，默认为 default 译码器
Mapping	可选	string	该列的映射信息，这是某些译码器特有的
formatHint	可选	string	按照格式设置列译码器的起始位值
hidden	可选	boolean	在使用 DESCRIBE <table name>和 SELECT * 时隐藏该字段，默认是 false
comment	可选	string	添加列的备注信息，当使用 DESCRIBE <table name>会显示该信息

对 key 或者 message 的字段描述是没有数量限制的。

11.1.6 行解码

对于 key 和 message，译码器用来将 key 和 message 的数据映射到对应的列。如果表不存在对应的表定义文件，则 dummy 译码器将对数据进行解析。

Kafka Connector 包含如下译码器：

- raw: 不要转换行数据，使用原始字节数据。
- csv: 将数据解析为以逗号分隔的值。
- json: 将数据解析为 JSON 对象。

解码器的主要目的是选择适当的字段解码器来解释 message 或 key 的数据。

Presto 只支持 boolean、long、double 和 bytes 序列（被当作 string）这 4 种数据类型来和 Presto 的数据类型进行映射。

1. RAW 译码器

RAW 译码器支持从 message 或者 key 中读取原始数值（基于字节），然后将其转换为 Presto 中的列。

对于 fields，其支持以下属性：

- type: 所有的 Presto 类型都支持。
- dataformat: 只支持_default，可以省略。
- mapping: 选择被转换的数据类型的宽度。
- formatHint: 可选的，格式为<start>[:<end>]，表示被转换字节的起始和结束位置。

其中，属性 mapping 用来选择需要转换的字节数，在省略的情况下，默认为 BYTE。各个类型的值都支持。mapping 属性支持的数据类型有：

- BYTE: 1 个字节。
- SHORT: 2 个字节。
- INT: 4 个字节。
- LONG: 8 个字节。
- FLOAT: 4 个字节（IEEE 754 格式）。
- DOUBLE: 8 个字节（IEEE 754 格式）。

type 列定义了数值被转换成的 Presto 数据类型。

- 基于 boolean 的类型需要与 BYTE、SHORT、INT 或者 LONG 相对应。如果是其他类型，则会抛出转换错误。数值 0 返回 false，其他值返回 true。
- 基于 long 的类型需要和 BYTE、SHORT、INT 或者 LONG 对应。其他类型会抛出转换错误。
- 基于 double 类型需要和 FLOAT 或者 DOUBLE 对应。其他类型会抛出转换错误。
- 基于 string 的类型需要和 BYTE 对应。其他类型会抛出转换错误。

formatHint 字段指定了 key 或 message 中需要被转换的字节的位置。它可以是一个数字

或者两个冒号隔开的数字 (<start>[:<end>])。如果只有一个起始位置，则对应的字段会根据数据类型选取适当的字节数（如上所述）。基于 `string` 的类型（`VARCHAR`）会将开始到结束所有的字节进行转换。如果给定了开始和结束的位置，则所给定的字节数必须不少于对应类型规定的字节数。基于字符串的类型，开始位置（包括）和结束位置（不包括）之间的所有字节都会被转换。

2. CSV 译码器

CSV 译码器将 `message` 或 `key` 的字节数据转换为 UTF-8 编码的字符串，然后将结果解析为一行 CSV（逗号分隔的值）。

对于 `fields`，其支持以下属性：

- `type`: 支持 Presto 所有类型。
- `dataFormat`: 只支持 `_default`，可以省略。
- `mapping`: 列的字段索引，必须给定。
- `formatHint`: 不支持，忽略。
- 如果 `type` 为 `boolean` 类型，且 `field` 的值为 “true”（不区分大小写），则返回 `true`，否则返回 `false`。
- 如果 `type` 为 `long` 和 `double` 类型，则会根据 Java 对 `long` 和 `double` 类型的解析规则来对 `field` 的值进行解析。
- 如果 `type` 为字符串类型，则会将 `field` 的值原样输出（UTF-8 编码的文本）。

3. JSON 译码器

JSON 译码器根据 RFC 4627 规范将 `message` 或 `key` 的字节数据转换为 JSON 对象。注意，`message` 或 `key` 必须转换为一个 JSON 对象，而不能是一个数组或简单类型。

对于 `fields`，其支持以下属性：

- `type`: 支持 Presto 所有类型。
- `dataFormat`: 支持 `_default`、`custom-date-time`、`iso8601`、`rfc2822`、`milliseconds-since-epoch`、`seconds-since-epoch`。如果省略，则默认为 `_default`。
- `mapping`: 使用斜线分隔的字段名从 JSON 对象中选出特定的字段。
- `formatHint`: 只支持 `custom-date-time`。

JSON 译码器支持多个 `field` 译码器，对于标准的表列使用 `_default` 译码器，还有针对 `date` 和 `time` 类型的一些译码器。

4. `_default` Field 译码器

这是标准的字段译码器，支持 Presto 所有的数据类型。一个字段的数值会被 JSON 转换规则强制转换为 `boolean`、`long`、`double` 或 `string` 类型数值。对于非 `date` 或 `time` 类型的列，应使用该译码器。

11.1.7 日期和时间解码器

将 JSON 对象中的数值转换为 Presto 的 DATE、TIME 或者 TIMESTAMP 列，通过使用字段定义中的 dateFormat 属性可以选择特殊的译码器。

11.1.8 文本解码器

- iso8601: 基于文本的译码器，将一个文本字段解析为一个 ISO 8601 的时间戳。
- rfc2822: 基于文本的译码器，将一个文本字段解析为一个 RFC 2822 的时间戳。
- custom-date-time: 基于文本的译码器，需要一个格式化的 hint 来将对应的文本解析为一个 Joda-Time 格式的字符串。

文本解码器支持的类型及解码后对应的类型如表 11-6 所示。

表 11-6		
Presto 类型	JSON Text	JSON Long
string 类型	不予改变	根据格式类型进行解析，返回从 1970-01-01 00:00:00.000 到现在的毫秒数
long-based 类型	根据格式类型进行解析，返回从 1970-01-01 00:00:00.000 到现在的毫秒数	作为毫秒数返回

11.1.9 数值解码器

- milliseconds-since-epoch: 基于数字的，把一个文本或数字解析为从 1970-01-01 00:00:00.000 到现在的毫秒数。
- seconds-since-epoch: 基于数字的，把一个文本形式的数值或数字解析为从 1970-01-01 00:00:00.000 到现在的毫秒数。

数值解码器支持的类型及解码后对应的类型如表 11-7 所示。

表 11-7		
Presto 类型	JSON Text	JSON Long
string 类型	解析为 ISO8601 格式的 long 型数值	转成 ISO 8601 格式
long-based 类型	解析为 long 型，返回从 1970-01-01 00:00:00.000 到现在的毫秒数	返回从 1970-01-01 00:00:00.000 到现在的毫秒数

11.2 Kafka 连接器使用教程

Presto 的 Kafka Connector 允许通过 Presto 从 Apache Kafka 中获取实时的 topic 数据。本节讲述怎么设置 topic 和怎么创建表定义文件，从而返回对应的 Presto 表数据。

本节内容基于读者已经熟悉 Presto 并且知道如何部署 Presto。本节重点放在搭建 Kafka 集群和将它与 Presto 进行集成。

11.2.1 安装 Apache Kafka

本节内容基于 kafka-0.8.2.1, 我们可以从 Apache Kafka 官方网站进行下载。当前的 Kafka Connector 支持 Apache Kafka 所有 0.8.X 系列版本或更高版本。

下载 Apache Kafka 的二进制包并解压, Apache Kafka 自带了 zookeeper, 可以直接使用。启动 Zookeeper 和 Kafka server:

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
[2015-09-14 15:49:15,566] INFO Reading configuration from:
config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
...
```

```
$ bin/kafka-server-start.sh config/server.properties
[2015-09-14 16:03:31,803] INFO Verifying properties
(kafka.utils.VerifiableProperties)
[2015-09-14 16:03:31,832] INFO Property broker.id is overridden to 0
(kafka.utils.VerifiableProperties)
...
```

Zookeeper 默认的 ClientPort 是 2181, Kafka 的 ClientPort 是 9092。

11.2.2 下载数据

从 maven 中心仓库下载 tpch-kafka loader:

```
$ curl -o kafka-tpch
https://repol.maven.org/maven2/de/softwareforge/kafka_tpch_0811/1.0/kafka_tpch_0811-1.0.sh
$ chmod 755 kafka-tpch
```

现在运行 kafka-tpch 程序来预加载一些 topics, 这些 topics 里面有 tpch 数据。

```
$ ./kafka-tpch load --brokers localhost:9092 --prefix tpch. --tpch-type tiny
2015-09-14T16:22:41.876+0800 INFO main io.airlift.log.Logging Logging
to stderr
2015-09-14T16:22:41.901+0800 INFO main
de.softwareforge.kafka.LoadCommand Processing tables: [customer, orders,
lineitem, part, partsupp, supplier, nation, region]
2015-09-14T16:22:42.293+0800 INFO pool-1-thread-1
de.softwareforge.kafka.LoadCommand Loading table 'customer' into topic
'tpch.customer'...
2015-09-14T16:22:42.293+0800 INFO pool-1-thread-2
de.softwareforge.kafka.LoadCommand Loading table 'orders' into topic
'tpch.orders'...
2015-09-14T16:22:42.294+0800 INFO pool-1-thread-3
de.softwareforge.kafka.LoadCommand Loading table 'lineitem' into topic
'tpch.lineitem'...
2015-09-14T16:22:42.295+0800 INFO pool-1-thread-4
de.softwareforge.kafka.LoadCommand Loading table 'part' into topic
'tpch.part'...
2015-09-14T16:22:42.296+0800 INFO pool-1-thread-5
```

```

de.softwareforge.kafka.LoadCommand      Loading table 'partsupp' into topic
'tpch.partsupp'...
2015-09-14T16:22:42.296+0800      INFO      pool-1-thread-6
de.softwareforge.kafka.LoadCommand      Loading table 'supplier' into topic
'tpch.supplier'...
2015-09-14T16:22:42.297+0800      INFO      pool-1-thread-7
de.softwareforge.kafka.LoadCommand      Loading table 'nation' into topic
'tpch.nation'...
2015-09-14T16:22:42.298+0800      INFO      pool-1-thread-8
de.softwareforge.kafka.LoadCommand      Loading table 'region' into topic
'tpch.region'...
2015-09-14T16:22:53.295+0800      ERROR      pool-1-thread-8
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.region
2015-09-14T16:22:53.456+0800      INFO      pool-1-thread-8
de.softwareforge.kafka.LoadCommand      Generated 5 rows for table 'region'.
2015-09-14T16:22:53.478+0800      ERROR      pool-1-thread-1
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.customer
2015-09-14T16:22:53.622+0800      ERROR      pool-1-thread-3
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.lineitem
2015-09-14T16:22:53.756+0800      ERROR      pool-1-thread-2
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.orders
2015-09-14T16:22:53.883+0800      ERROR      pool-1-thread-6
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.supplier
2015-09-14T16:22:53.997+0800      INFO      pool-1-thread-6
de.softwareforge.kafka.LoadCommand      Generated 100 rows for table 'supplier'.
2015-09-14T16:22:54.006+0800      ERROR      pool-1-thread-4
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.part
2015-09-14T16:22:54.134+0800      ERROR      pool-1-thread-5
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.partsupp
2015-09-14T16:22:54.251+0800      ERROR      pool-1-thread-7
kafka.producer.async.DefaultEventHandler      Failed to collate messages by topic,
partition due to: Failed to fetch topic metadata for topic: tpch.nation
2015-09-14T16:22:54.359+0800      INFO      pool-1-thread-7
de.softwareforge.kafka.LoadCommand      Generated 25 rows for table 'nation'.
2015-09-14T16:22:55.295+0800      INFO      pool-1-thread-1
de.softwareforge.kafka.LoadCommand      Generated 1500 rows for table 'customer'.
2015-09-14T16:22:55.501+0800      INFO      pool-1-thread-4
de.softwareforge.kafka.LoadCommand      Generated 2000 rows for table 'part'.
2015-09-14T16:22:56.339+0800      INFO      pool-1-thread-5
de.softwareforge.kafka.LoadCommand      Generated 8000 rows for table 'partsupp'.
2015-09-14T16:22:56.455+0800      INFO      pool-1-thread-2
de.softwareforge.kafka.LoadCommand      Generated 15000 rows for table 'orders'.
2015-09-14T16:22:58.119+0800      INFO      pool-1-thread-3
de.softwareforge.kafka.LoadCommand      Generated 60175 rows for table 'lineitem'.

```

现在，Kafka 中已经有 topics 并且这些 topics 中已经预加载了一些数据可以用来查询。

11.2.3 在 Presto 中配置 Kafka topics

在 Presto 安装路径中，为 Kafka Connector 增加一个 Catalog 的属性文件：
etc/catalog/kafka.properties。该文件列出了 Kafka 的节点和 topics：

```
connector.name=kafka
kafka.nodes=localhost:9092
kafka.table-names=tpch.customer,tpch.orders,tpch.lineitem,tpch.part,tpch.part
supp,tpch.supplier,tpch.nation,tpch.region
kafka.hide-internal-columns-hidden=false
```

然后启动 Presto：

```
$ bin/launcher start
```

因为在配置文件中，Kafka 中的表都带有前缀 tpch，这些表的 Schema 就是 tpch。因为属性文件的名称是 kafka.properties，所以对应的 Catalog 的名称就是 Kafka。

启动 Presto Cli：

```
$ ./presto --catalog kafka --schema tpch
```

列出表确认配置能够生效：

```
presto:tpch> SHOW TABLES;
Table
-----
customer
lineitem
nation
orders
part
partsupp
region
supplier
(8 rows)
```

11.2.4 基本数据查询

Kafka 中的数据是非结构化的，没有元数据来描述消息的格式。如果没有进一步的配置，则 Kafka Connector 可以访问数据并以原始的数据格式展示，展示的数据只有内置的列而没有实际列。

```
presto:tpch> DESCRIBE customer;
Column      | Type   | Null | Partition Key | Comment
-----+-----+-----+-----+-----
partition_id | bigint | true  | false         | Partition Id
```



```

_partition_offset | bigint | true | false | Offset for the message within the
partition
_segment_start   | bigint | true | false | Segment start offset
_segment_end     | bigint | true | false | Segment end offset
_segment_count   | bigint | true | false | Running message count per segment
_key             | varchar | true | false | Key text
_key_corrupt     | boolean | true | false | Key data is corrupt
_key_length      | bigint | true | false | Total number of key bytes
_message         | varchar | true | false | Message text
_message_corrupt | boolean | true | false | Message data is corrupt
_message_length  | bigint | true | false | Total number of message bytes
(11 rows)

presto:tpch> SELECT count(*) FROM customer;
_col0
-----
1500

presto:tpch> SELECT _message FROM customer LIMIT 5;
_message
-----
-----
-----
-----
-----

{"rowNumber":1,"customerKey":1,"name":"Customer#000000001","address":"IVhzIApeRb
ot,c,E","nationKey":15,"phone":"25-989-741-2988","accountBalance":711.56,"marketSe
gme

{"rowNumber":2,"customerKey":2,"name":"Customer#000000002","address":"XSTf4,NCwDVA
WNe6tEgvwfmRchLXak","nationKey":13,"phone":"23-768-687-3665","accountBalance":121.
65

{"rowNumber":3,"customerKey":3,"name":"Customer#000000003","address":"MG9kdTD2WBHm
","nationKey":1,"phone":"11-719-748-3364","accountBalance":7498.12,"marketSegment
":"

{"rowNumber":4,"customerKey":4,"name":"Customer#000000004","address":"XxVSJsLAGtn"
,"nationKey":4,"phone":"14-128-190-5944","accountBalance":2866.83,"marketSegment":
"M

{"rowNumber":5,"customerKey":5,"name":"Customer#000000005","address":"KvpyuHCplrB8
4WgAiGV6sYpZq7Tj","nationKey":3,"phone":"13-750-942-6364","accountBalance":794.47,
"m

(5 rows)

presto:tpch> SELECT sum(cast(json_extract_scalar(_message, '$.accountBalance')
AS double)) FROM customer LIMIT 10;
_col0
-----
6681865.590000002
```


(1 row)

Kafka 中的数据可以通过 Presto 进行查询，但是实际上它并没有表的结构。原始数据可以通过 `_message` 和 `_key` 列查询出来，但是它们并没有进一步被解析成更具体的列。由于样本数据是 JSON 格式的，可以使用 Presto 内置的 JSON 函数对数据进行进一步划分。

11.2.5 添加表定义文件

Kafka Connector 支持表定义文件以将原始数据映射成表的形式。这些文件位于 Presto 安装路径下的 `etc/kafka` 目录下面，并且必须以 `.json` 结尾。建议文件名与对应表名匹配，但这不是必需的。

添加如下文件 `etc/kafka/tpch.customer.json` 并重启 Presto:

```
{
  "tableName": "customer",
  "schemaName": "tpch",
  "topicName": "tpch.customer",
  "key": {
    "dataFormat": "raw",
    "fields": [
      {
        "name": "kafka_key",
        "dataFormat": "LONG",
        "type": "BIGINT",
        "hidden": "false"
      }
    ]
  }
}
```

现在表 `customer` 多了一个额外的列: `kafka_key`。

```
presto:tpch> DESCRIBE customer;
```

Column	Type	Null	Partition Key	Comment
kafka_key	bigint	true	false	
_partition_id	bigint	true	false	Partition Id
_partition_offset	bigint	true	false	Offset for the message within the partition
_segment_start	bigint	true	false	Segment start offset
_segment_end	bigint	true	false	Segment end offset
_segment_count	bigint	true	false	Running message count per segment
_key	varchar	true	false	Key text
_key_corrupt	boolean	true	false	Key data is corrupt
_key_length	bigint	true	false	Total number of key bytes
_message	varchar	true	false	Message text
_message_corrupt	boolean	true	false	Message data is corrupt

```
_message_length      | bigint | true   | false | Total number of message bytes
(12 rows)

presto:tpch> SELECT kafka_key FROM customer ORDER BY kafka_key LIMIT 10;
kafka_key
-----
0
1
2
3
4
5
6
7
8
9
(10 rows)
```

Topic 定义文件将 Kafka 内部的 key（在该 topic 中是 8 字节 long 型数据）映射为 Presto 中类型为 BIGINT 的列，列名为 kafka_key。

11.2.6 将 message 中所有值映射到不同列

更新文件 etc/kafka/tpch.customer.json，为 message 增加字段并重启 Presto。因为 message 中的字段都是 JSON 格式，所以 message 的 dataForma 为 json。这是一个 key 和 message 分别使用不同数据格式的例子。

```
{
  "tableName": "customer",
  "schemaName": "tpch",
  "topicName": "tpch.customer",
  "key": {
    "dataFormat": "raw",
    "fields": [
      {
        "name": "kafka_key",
        "dataFormat": "LONG",
        "type": "BIGINT",
        "hidden": "false"
      }
    ]
  },
  "message": {
    "dataFormat": "json",
    "fields": [
      {
        "name": "row_number",
        "mapping": "rowNumber",
        "type": "BIGINT"
      }
    ]
  }
}
```

```

        "name": "customer_key",
        "mapping": "customerKey",
        "type": "BIGINT"
    },
    {
        "name": "name",
        "mapping": "name",
        "type": "VARCHAR"
    },
    {
        "name": "address",
        "mapping": "address",
        "type": "VARCHAR"
    },
    {
        "name": "nation_key",
        "mapping": "nationKey",
        "type": "BIGINT"
    },
    {
        "name": "phone",
        "mapping": "phone",
        "type": "VARCHAR"
    },
    {
        "name": "account_balance",
        "mapping": "accountBalance",
        "type": "DOUBLE"
    },
    {
        "name": "market_segment",
        "mapping": "marketSegment",
        "type": "VARCHAR"
    },
    {
        "name": "comment",
        "mapping": "comment",
        "type": "VARCHAR"
    }
]
}

```

现在对于 message 中所有 JSON 格式的字段，都定义了对应的列。对于之前求和的查询操作可以直接通过对 account_balance 列进行操作来完成。

```
presto:tpch> DESCRIBE customer;
```

Column	Type	Null	Partition Key	Comment
kafka_key	bigint	true	false	
row_number	bigint	true	false	

customer_key	bigint	true	false	
name	varchar	true	false	
address	varchar	true	false	
nation_key	bigint	true	false	
phone	varchar	true	false	
account_balance	double	true	false	
market_segment	varchar	true	false	
comment	varchar	true	false	
_partition_id	bigint	true	false	Partition Id
_partition_offset	bigint	true	false	Offset for the message within the partition
_segment_start	bigint	true	false	Segment start offset
_segment_end	bigint	true	false	Segment end offset
_segment_count	bigint	true	false	Running message count per segment
_key	varchar	true	false	Key text
_key_corrupt	boolean	true	false	Key data is corrupt
_key_length	bigint	true	false	Total number of key bytes
_message	varchar	true	false	Message text
_message_corrupt	boolean	true	false	Message data is corrupt
_message_length	bigint	true	false	Total number of message bytes

(21 rows)

```
presto:tpch> SELECT * FROM customer LIMIT 5;
```

kafka_key	row_number	customer_key	name	address
nation_key	phone	account_balance	market_segment	
0	1	1	Customer#000000001	IVhzIApeRb ot,c,E
15	25-989-741-2988	711.56	BUILDING	to the
1	2	2	Customer#000000002	
XSTf4,NCwDVaWNe6tEgvwfmRchLXak.	13	23-768-687-3665		121.65
AUTOMOBILE	1	acco		
2	3	3	Customer#000000003	MG9kdTD2WBHm
1	11-719-748-3364	7498.12	AUTOMOBILE	depos
3	4	4	Customer#000000004	XxVSJsLAGtn
4	14-128-190-5944	2866.83	MACHINERY	reque
4	5	5	Customer#000000005	
KvpyuHCplrB84WgAiGV6sYpZq7Tj	3	13-750-942-6364		794.47
HOUSEHOLD	1	n	acco	

(5 rows)

```
presto:tpch> SELECT sum(account_balance) FROM customer LIMIT 10;
```

_col0
6681865.5900000002

(1 row)

11.2.7 使用实时数据

当数据到达 Kafka 时，Presto 即可将数据实时查询出来。本节模拟实时数据查询。

1. 创建生产者向 Kafka 发送数据

首先我们在 Kafka 集群中新建一个 topic: jd.weblog

```
$ bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic jd.weblog
```

根据 Apache Kafka 的 Java API, 编写 Producer 程序:

```
import com.google.common.primitives.Ints;

import kafka.producer.Partitioner;
import kafka.utils.VerifiableProperties;

public class LongPartitioner implements Partitioner {
    public LongPartitioner(VerifiableProperties props) {
    }

    public int partition(Object value, int numPartitions) {
        if (value instanceof Long) {
            return Ints.checkedCast(((Long) value) %
numPartitions);
        } else {
            return 0;
        }
    }
}

import kafka.serializer.Encoder;
import kafka.utils.VerifiableProperties;

import java.nio.ByteBuffer;

public class LongEncoder implements Encoder<Long> {

    public LongEncoder(VerifiableProperties props) {
    }

    public byte[] toBytes(Long key) {
        ByteBuffer buf = ByteBuffer.allocate(8);
        buf.putLong(key);
        return buf.array();
    }
}

import java.util.Date;
import java.util.Properties;
import java.util.Random;

import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
import kafka.serializer.StringEncoder;
```

```

public class TestProducer {

    public static void main(String[] args) {

        Properties props = new Properties();
        props.put("metadata.broker.list",
            "localhost:9092");
        props.put("serializer.class", StringEncoder.class.getName());
        props.put("key.serializer.class", LongEncoder.class.
getName());

        props.put("partitioner.class", LongPartitioner.class.
getName());

        props.put("serializer.encoding", "UTF8");
        props.put("request.required.acks", "1");

        ProducerConfig config = new ProducerConfig(props);
        Producer<Long, String> producer = new Producer<Long, String>
(config);

        String[] names = { "Alice", "Bob", "Cara", "David", "Edward" };
        String[] genders = { "female", "male", "female", "male",
"male" };

        String[] websites = { "www.jd.com", "www.google.com",
"www.facebook.com", "www.linkedin.com", "www.twitter.com" };
        Random rnd = new Random();
        long count = 0;
        while (true) {

            StringBuffer sb = new StringBuffer("{ \"user\":{ \"name\":\" \";
int number = rnd.nextInt(5);

            sb.append(names[number]);
            sb.append("\", \"gender\": \"\");
            sb.append(genders[number]);
            sb.append("\", \"time\": \");
            long runtime = new Date().getTime();
            sb.append(runtime);
            sb.append("\", \"website\": \"\");
            number = rnd.nextInt(5);
            sb.append(websites[number]);
            sb.append("\")");

            String msg = sb.toString();
            KeyedMessage<Long, String> data = new KeyedMessage<Long,
String>(
                "jd.weblog", count++, msg);
            producer.send(data);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
  }
}

```

将编译打包，在集群上执行：

```
$ java -jar tpchProducer.jar com.jd.kafka.tpchproducer.TestProducer
```

2. 在 Presto 中创建表

编辑 Presto 安装目录下的 `etc/catalog/kafka.properties` 文件，在属性 `kafka.table-names` 中添加 `jd.weblog`

添加 topic 定义文件，在 Presto 安装目录下的 `/etc/kafka/` 目录内添加 `jd.weblog.json` 文件，文件内容为：

```

{
  "tableName": "weblog",
  "schemaName": "jd",
  "topicName": "jd.weblog",
  "key": {
    "dataFormat": "raw",
    "fields": [
      {
        "name": "kafka_key",
        "dataFormat": "LONG",
        "type": "BIGINT",
        "hidden": "false"
      }
    ]
  },
  "message": {
    "dataFormat": "json",
    "fields": [
      {
        "name": "name",
        "mapping": "user/name",
        "type": "VARCHAR"
      },
      {
        "name": "gender",
        "mapping": "user/gender",
        "type": "VARCHAR"
      },
      {
        "name": "time",
        "mapping": "time",
        "type": "TIMESTAMP"
      },
      {
        "name": "website",
        "mapping": "website",

```



```
        "type": "VARCHAR"
      }
    }
  }
}
```

重新启动 Presto，然后启动 Presto CLI:

```
$ ./presto --catalog kafka --schema jd
```

通过查询语句看是否已经生效:

```
presto:jd> show tables;
Table
-----
weblog
(1 rows)
```

查询一下表信息:

```
presto:jd> desc weblog;
Column      | Type      | Null | Partition Key | Comment
-----+-----+-----+-----+-----
kafka_key   | bigint    | true | false          |
name        | varchar   | true | false          |
gender      | varchar   | true | false          |
time        | timestamp | true | false          |
website     | varchar   | true | false          |
(5 rows)
```

和 topic 定义文件中配置相一致。

3. 在 Presto 中查询数据

由于我们的生产者程序一直在运行，所以 Kafka 的 `jd.weblog` 下的数据会逐渐增多，而 Presto 可以实时查询 Kafka 中到达的数据，因此，如果对表 `weblog` 中的数据进行统计，则会发现数据逐渐增多。

```
presto:jd> select count(*) from weblog;
 _col0
-----
   767
(1 row)

presto:jd> select count(*) from weblog;
 _col0
-----
   777
(1 row)

presto:jd> select count(*) from weblog;
 _col0
-----
   959
```


(1 row)

presto:jd> select name, gender, time, website from weblog limit 10;

name	gender	time	website
Bob	male	2015-09-15 16:14:55.453	www.twitter.com
Edward	male	2015-09-15 16:14:58.483	www.google.com
Bob	male	2015-09-15 16:15:01.492	www.twitter.com
David	male	2015-09-15 16:15:04.500	www.twitter.com
David	male	2015-09-15 16:15:07.508	www.google.com
Alice	female	2015-09-15 16:15:10.515	www.facebook.com
Bob	male	2015-09-15 16:15:13.529	www.facebook.com
Alice	female	2015-09-15 16:15:16.537	www.google.com
Edward	male	2015-09-15 16:15:19.544	www.facebook.com
Alice	female	2015-09-15 16:15:22.552	www.facebook.com

(10 rows)

11.3 Kafka Connector 获取数据

11.3.1 Split 分片管理

Kafka Connector 获取 Kafka 中的分片信息是在类 `KafkaSplitManager` 的 `getPartitionSplits()` 方法中实现的。在该方法中，对 topic 的所有分区进行遍历，对每个分区获取对应的 leader 节点并通过 `getConsumer()` 方法获取该 leader 节点的 `SimpleConsumer` 对象。然后通过 `findAllOffsets()` 方法获取分区内所有 segment 的起始偏移量（即该 segment file 的文件名），之后将每个 segment 封装成一个 Split。

```
public ConnectorSplitSource getPartitionSplits(ConnectorTableHandle tableHandle,
List<ConnectorPartition> partitions)
{
    KafkaTableHandle kafkaTableHandle = handleResolver.convertTableHandle
(tableHandle);

    ImmutableList.Builder<ConnectorSplit> builder = ImmutableList.builder();

    //对分区进行遍历
    for (ConnectorPartition cp : partitions) {
        checkState(cp instanceof KafkaPartition, "Found an unknown partition
type: %s", cp.getClass().getSimpleName());
        KafkaPartition partition = (KafkaPartition) cp;

        //获取分区的 leader 节点，并获取对应 SimpleConsumer 对象
        SimpleConsumer leaderConsumer = consumerManager.getConsumer(partition.
getPartitionLeader());
        // Kafka contains a reverse list of "end - start" pairs for the splits
        //获取该分区下所有 Segment 偏移量的集合
        long[] offsets = findAllOffsets(leaderConsumer, partition);
        //每个 segment 封装成一个 split
        for (int i = offsets.length - 1; i > 0; i--) {
```

```

        KafkaSplit split = new KafkaSplit(connectorId,
            partition.getTopicName(),
            kafkaTableHandle.getKeyDataFormat(),
            kafkaTableHandle.getMessageDataFormat(),
            partition.getPartitionIdAsInt(),
            offsets[i],
            offsets[i - 1],
            partition.getPartitionNodes());
        builder.add(split);
    }

    return new FixedSplitSource(connectorId, builder.build());
}

```

下面看一下获取分区内所有 segment 起始偏移量的原理：它是由 `findAllOffsets()` 方法实现的。方法体内调用 Kafka API 中的 `PartitionOffsetRequestInfo` 类，该类的构造方法有两个参数，第一个是 `Long` 型的时间戳，表示要获取最后修改时间小于该时间戳的 segment 的起始偏移量，这里的取值为系统当前时间。第二个参数是 `int` 型的 `offset` 数组的长度，这里值为 `Integer.MAX_VALUE`，实际取值为修改时间小于当前时间的 segment 的数量与 `Integer.MAX_VALUE` 这两个值中的最小值，也就是返回该分区内所有 segment 的起始偏移量。

```

private static long[] findAllOffsets(SimpleConsumer consumer, KafkaPartition
partition)
{
    TopicAndPartition topicAndPartition = new
TopicAndPartition(partition.getTopicName(), partition.getPartitionIdAsInt());

    // The API implies that this will always return all of the offsets. So it
seems a partition can not have
    // more than Integer.MAX_VALUE-1 segments.
    //
    // This also assumes that the lowest value returned will be the first segment
available. So if segments have been dropped off, this value
    // should not be 0.
    PartitionOffsetRequestInfo partitionOffsetRequestInfo = new
PartitionOffsetRequestInfo(kafka.api.OffsetRequest.LatestTime(),
Integer.MAX_VALUE);
    OffsetRequest offsetRequest = new OffsetRequest(ImmutableMap.
of(topicAndPartition, partitionOffsetRequestInfo),
kafka.api.OffsetRequest.CurrentVersion(), consumer.clientId());
    OffsetResponse offsetResponse = consumer.getOffsetsBefore
(offsetRequest);

    if (offsetResponse.hasError()) {
        short errorCode = offsetResponse.errorCode(partition.getTopicName(),
partition.getPartitionIdAsInt());
        log.warn("Offset response has error: %d", errorCode);
        throw new PrestoException(KAFKA_SPLIT_ERROR, "could not fetch data from

```

```
Kafka, error code is '" + errorCode + "'");
    }

    return offsetResponse.offsets(partition.getTopicName(),
partition.getPartitionIdAsInt());
}
```

11.3.2 数据读取

获取 topic 的分片信息后就可以去 Kafka 中获取数据了，数据的获取主要通过 `KafkaRecordSet` 类来实现。通过 `KafkaRecordSet` 类中的 `cursor()` 方法来实现。

```
public RecordCursor cursor()
{
    return new KafkaRecordCursor();
}
```

该方法又调用了 `KafkaRecordSet` 的内部类 `KafkaRecordCursor`。该类中一个重要的方法是：

```
public boolean advanceNextPosition()
{
    while (true) {
        if (cursorOffset >= split.getEnd()) {
            return endOfData(); // Split end is exclusive.
        }
        // Create a fetch request
        openFetchRequest();

        while (messageAndOffsetIterator.hasNext()) {
            MessageAndOffset currentMessageAndOffset =
messageAndOffsetIterator.next();
            long messageOffset = currentMessageAndOffset.offset();

            if (messageOffset >= split.getEnd()) {
                return endOfData(); // Past our split end. Bail.
            }

            if (messageOffset >= cursorOffset) {
                return nextRow(currentMessageAndOffset);
            }
        }
        messageAndOffsetIterator = null;
    }
}
```

该方法用来获取 Split 的数据，当 `cursorOffset` 小于 `split.getEnd()` 时，调用 `openFetchRequest()` 方法，将获取的数据赋值给 `messageAndOffsetIterator`。然后从迭代器 `messageAndOffsetIterator` 中获取 `offset` 来判断该 Split 的数据是否获取完成。

```
private void openFetchRequest()
{
}
```



```

        if (messageAndOffsetIterator == null) {
            log.debug("Fetching %d bytes from offset %d (%d - %d). %d messages
read so far", KAFKA_READ_BUFFER_SIZE, cursorOffset, split.getStart(), split.getEnd(),
totalMessages);

            FetchRequest req = new FetchRequestBuilder()
                .clientId("presto-worker-" + Thread.currentThread().
getName())
                .addFetch(split.getTopicName(), split.getPartitionId(),
cursorOffset, KAFKA_READ_BUFFER_SIZE)
                .build();

            // TODO - this should look at the actual node this is running on and
prefer
            // that copy if running locally. - look into NodeInfo
            SimpleConsumer consumer =
consumerManager.getConsumer(split.getNodes().get(0));

            FetchResponse fetchResponse = consumer.fetch(req);
            if (fetchResponse.hasError()) {
                short errorCode = fetchResponse.errorCode(split.getTopicName(),
split.getPartitionId());
                log.warn("Fetch response has error: %d", errorCode);
                throw new PrestoException(KAFKA_SPLIT_ERROR, "could not fetch
data from Kafka, error code is '" + errorCode + "'");
            }

            messageAndOffsetIterator =
fetchResponse.messageSet(split.getTopicName(), split.getPartitionId()).iterator();
        }
    }
}

```

`openFetchRequest()`方法通过调用 Kafka 的 API,通过创建 `FetchRequest`、`SimpleConsumer` 和 `FetchResponse` 类的实例来从 Kafka 中获取数据,将获取到的 Split 所在 partition 的数据通过调用 `iterator()`方法赋值给迭代器 `messageAndOffsetIterator`。

11.4 小结

本章详细描述了 Kafka Connector, 主要介绍了 Kafka Connector 的配置, 对各种配置参数, 以及对表的定义文件进行了介绍。然后重点讲解了 Kafka Connector 的具体使用, 包括 Apache Kafka 的安装、在 Presto 中配置 Kafka topics、数据的查询、表描述文件的添加以及实时数据的获取。最后对 Kafka Connector 的源代码进行了简单讲解, 包括 Split 分片管理和数据获取的代码实现。

第12章

Connector 开发

Presto 可以支持很多的数据源，并且数据源都是以 plugin 的方式添加的，Presto 整个工程编译完成后，会形成如下的目录结构：

```
Presto-server-0.107
|-----bin
|-----lib
|-----Plugin
|           |-----cassandra
|           |-----example-http
|           |-----hive-cdh4
|           |-----hive-cdh5
|           |-----hive-hadoop1
|           |-----hive-hadoop2
|           |-----kafka
|           |-----ml
|           |-----mysql
|           |-----postgresql
|           |-----raptor
|           |-----tpch
```

Plugin 目录包含了所有的 Connector，每个 Plugin 目录下包含了各自所需要的 jar 包，而公共 jar 包可以放到上层的 lib 目录下。

Presto 定义了非常友好的接口提供给广大开发者，以便添加新的 Connector，本章以 presto-example-http 为例，详细介绍在 Presto 中如何添加一个新的 Connector。

12.1 创建 Maven 工程

在 Presto 目录下创建一个 presto-example-http 的目录，在 presto-example-http 根目录下编辑 pom.xml 配置文件，以下几项是必须配置的。

- 将当前工程加入到 Presto 组中，版本需要和整个 Presto 工程的版本一致。

```
<parent>
  <groupId>com.facebook.presto</groupId>
  <artifactId>presto-root</artifactId>
  <version>0.107</version>
</parent>
```

- 描述当前工程的信息，`packaging` 选项为 `presto-plugin`，在打包编译时，会将当前工程打包编译到 Presto 的 `plugin` 目录下。

```
<artifactId>presto-example-http</artifactId>
<description>Presto - Example HTTP Connector</description>
<packaging>presto-plugin</packaging>
```

- 每个 Plugin 工程都需要依赖 `presto-spi` 模块，`presto-spi` 中统一定义了各种 API 接口。

```
<dependency>
  <groupId>com.facebook.presto</groupId>
  <artifactId>presto-spi</artifactId>
  <scope>provided</scope>
</dependency>
```

其中 `scope` 选项的 `provided` 代表当前依赖的 jar 已经提供，`presto-spi` 最终会打包编译到 `presto/lib` 目录下，所有的 `plugin` 模块都会加载 `lib` 中的 jar，所以当前的 `presto-spi` 不需要重复打包编译到当前的 `plugin` 目录中。

在整个 Presto 工程编译时，需要将我们自定义添加的 `connector` 也编译打包到 `plugin` 目录下，通过以下的配置即可达到我们的目的。

- 在 Presto 根目录下的 `pom.xml` 中，将我们新增的 `Connector` 加入到 `modules` 中：

```
<modules>
  .....
  <module>presto-example-http</module>
  .....
</modules>
```

- 在 `presto-server` 的 `src/main/provisio/presto.xml` 配置文件中，注册新增的 `Connector`：

```
<artifactSet to="plugin/example-http">
  <artifact id="${project.groupId}:presto-example-http:zip:${project.
version}">
    <unpack />
  </artifact>
</artifactSet>
```

经过以上步骤，新增 `Connector` 的准备工作就已经完成。在开发过程中根据具体实现使用到的类来添加其依赖。Plugin 使用了独立的类加载器，和其他的类是隔离的，因此 Plugin 可以使用不同版本的类库，区别于 Presto 内部使用的版本，例如当前 Plugin 要使用的 jar 包与 Presto 内部所使用的版本有冲突，那么 Plugin 内部可以定义声明不同于 Presto 内部所使用的 jar 包版本。

12.2 注册 Plugin

每个 Plugin 必须要实现 `com.facebook.presto.spi.Plugin` 接口，Presto 会在服务启动环节加载所有的 Plugin，入口代码在 `com.facebook.presto.server.PrestoServer.run()` 方法中，其详细的代码为：

```
injector.getInstance(PluginManager.class).loadPlugins();
```

`injector` 为 Guice 的注解加载工具，其含义为：实例化 `PluginManager` 并且调用其 `loadPlugin` 方法。`loadPlugin` 方法的工作内容为：读取 Presto 部署的 Plugin 目录，构造类加载器，最终调用 `PluginManager.installPlugin(Plugin plugin)` 方法向 `PrestoServer` 注册 Plugin，其实现细节如以下代码所示：

```
for (ConnectorFactory connectorFactory : plugin.getServices(ConnectorFactory.class)) {
    log.info("Registering connector %s", connectorFactory.getName());
    connectorManager.addConnectorFactory(connectorFactory);
}
```

从上面我们可以看到，最终需要调用 Plugin 的 `getServices` 方法注册 connector，`getServices` 会返回一个 `ConnectorFactory` 列表，并且将该 `ConnectorFactory` 加入到 `ConnectorManager` 中，那么在注册一个 Connector 时需要 overwrite `getServices` 方法，在 `presto-example-http` module 的 `ExamplePlugin` 类中进行了实现，代码如下：

```
@Override
public synchronized <T> List<T> getServices(Class<T> type)
{
    if (type == ConnectorFactory.class) {
        return ImmutableList.of(type.cast(new ExampleConnectorFactory(
            typeManager, getOptionalConfig())));
    }
    return ImmutableList.of();
}
```

这样整个 Plugin 的注册工作就完成了。工厂类 `ExampleConnectorFactory` 继承自接口 `com.facebook.presto.spi.ConnectorFactory`（该接口属于 `presto-spi` 工程），因此 `ExampleConnectorFactory` 类中只有两个方法：

- `getName()`：返回 Plugin 的名称。
- `create(String connectorId, Map<String, String> config)`：注册该 Plugin 所要启动的 module，设置 Connector 的相关配置参数并完成 Connector 的实例化。

12.3 Connector

从 12.2 节我们得知，在注册 Plugin 时会将 `ConnectorFactory` 注册到 `ConnectorManager`，而且最终是需要通过 `ConnectorFactory.create` 方法来实例化 Connector，这个过程的发起点也

在 `PrestoServer.run` 方法中, 如以下代码所示:

```
injector.getInstance(CatalogManager.class).loadCatalogs();
```

使用 `CatalogManager` 加载 Presto 中配置的各个 `Catalog`, `loadCatalogs` 读取到 `Catalog` 的配置目录后, 依次加载各个 `Catalog`:

```
for (File file : listFiles(catalogConfigurationDir)) {
    if (file.isFile() && file.getName().endsWith(".properties")) {
        loadCatalog(file);
    }
}
```

进入到 `loadCatalog` 后, 最终根据 `catalogName` 获取到 `ConnectorFactory`, 在 `ConnectorManager.createConnection()` 方法中执行 `ConnectorFactory.create`:

```
public synchronized void createConnection(String catalogName, ConnectorFactory
connectorFactory, Map<String, String> properties)
{
    String connectorId = getConnectorId(catalogName);
    checkState(!connectors.containsKey(connectorId), "A connector %s already
exists", connectorId);
    Connector connector = connectorFactory.create(connectorId, properties);
    .....
}
```

`create` 方法由各个具体的 `ConnectorFactory` 实现, 也就是会调用自定义的 `ExampleConnectorFactory.create` 方法。该方法是为了构造一个 `Connector`, 利用 `Injector` 来创建 `ExampleModule` 并初始化相关配置。在 `ExampleModule` 的 `configure` 方法中, 绑定了 `Connector` 所需要的各个接口实现。

```
@Override
public void configure(Binder binder)
{
    // 数据源定义类
    binder.bind(ExampleConnector.class).in(Scopes.SINGLETON);
    // 数据源 ID
    binder.bind(ExampleConnectorId.class).toInstance(new ExampleConnectorId
(connectorId));
    // 数据源元数据的各个操作的入口类
    binder.bind(ExampleMetadata.class).in(Scopes.SINGLETON);
    // 数据源元数据的各个操作的实现类
    binder.bind(ExampleClient.class).in(Scopes.SINGLETON);
    // 数据源 split 管理类
    binder.bind(ExampleSplitManager.class).in(Scopes.SINGLETON);
    // 数据源数据读取类
    binder.bind(ExampleRecordSetProvider.class).in(Scopes.SINGLETON);
    // 数据源 handler 解析器
    binder.bind(ExampleHandleResolver.class).in(Scopes.SINGLETON);
    // 数据源相关配置文件
    bindConfig(binder).to(ExampleConfig.class);
    .....s
```



```

}

```

其中数据源连接器 `ExampleConnector` 定义了 4 个重要的接口实现类：

```

@Inject
public ExampleConnector(
    ExampleMetadata metadata,
    ExampleSplitManager splitManager,
    ExampleRecordSetProvider recordSetProvider,
    ExampleHandleResolver handleResolver)
{
    this.metadata = checkNotNull(metadata, "metadata is null");
    this.splitManager = checkNotNull(splitManager, "splitManager is null");
    this.recordSetProvider = checkNotNull(recordSetProvider, "recordSetProvider
is null");
    this.handleResolver = checkNotNull(handleResolver, "handleResolver is null");
}
@Override
public ConnectorMetadata getMetadata()
{
    // 返回元数据管理类
    return metadata;
}
@Override
public ConnectorSplitManager getSplitManager()
{
    // 返回 split 管理类
    return splitManager;
}
@Override
public ConnectorRecordSetProvider getRecordSetProvider()
{
    // 返回数据读取类
    return recordSetProvider;
}
@Override
public ConnectorHandleResolver getHandleResolver()
{
    // 返回 handler 解析器
    return handleResolver;
}

```

12.4 Metadata

`ConnectorMetadata` 提供了对数据源元数据的管理和各种操作，例如列举 Schemas、Tables、Columns 信息，创建表、添加分区等。自定义 Plugin Connector 都需要实现 `ConnectorMetadata` 的各个方法，presto-example-http 中的 `ConnectorMetadata` 对应的实现类为 `ExampleMetadata`。`ExampleMetadata` 为 `com.facebook.presto.spi.ReadOnlyConnectorMetadata` 的子类，`ReadOnlyConnectorMetadata` 本身实现了 `com.facebook.presto.spi.ConnectorMetadata`

接口，只是不支持 Create Table、Drop Table、Insert 等操作，其子类只实现了查询相关的方法。

若需要开发一种新的 Connector，则必须要重写 Metadata 中的以下几个方法。

- `public List<String> listSchemaNames(ConnectorSession session)`：列举当前 Catalog 下的 Schemas 列表，对应的是 `show schemas [from catalogName]` 语法。
- `public List<SchemaTableName> listTables(ConnectorSession session, String schemaNameOrNull)`：列举当前选择 `catalog.schema` 下的 Tables 信息，对应的是 `show tables [from catalogName.schemaName]` 语法。
- `public ExampleTableHandle getTableHandle(ConnectorSession session, SchemaTableName tableName)`：获取 SQL 中的表实例，`ExampleTableHandle` 实现了 `com.facebook.presto.spi.ConnectorTableHandle` 接口，在获取分片和表字段的信息时需要用到该实例。
- `public ConnectorTableMetadata getTableMetadata(ConnectorTableHandle table)`：获取表的元数据信息，主要包含了表字段信息、所在的 Schema、Owner 等。
- `public Map<SchemaTableName, List<ColumnMetadata>> listTableColumns(ConnectorSession session, SchemaTablePrefix prefix)`：获取当前 schema 下所有表的 Column 元数据信息。
- `public Map<String, ConnectorColumnHandle> getColumnHandles(ConnectorTableHandle tableHandle)`：获取表的列信息。
- `public ColumnMetadata getColumnMetadata(ConnectorTableHandle tableHandle, ConnectorColumnHandle columnHandle)`：获取表的某一列的元数据信息，包含了字段名称、类型等相关信息。

除以上列举的操作外，还有 `create/insert table/view` 对应的相关操作：

- `public void createTable(ConnectorSession session, ConnectorTableMetadata tableMetadata)`：创建一个表，`ConnectorTableMetadata` 中描述了所要创建表元数据信息，对应的是 `create table tableName(id bigint, name varchar)` 操作。
- `public void dropTable(ConnectorTableHandle tableHandle)`：删除表，对应的是 `drop table` 的操作。
- `public ConnectorOutputTableHandle beginCreateTable(ConnectorSession session, ConnectorTableMetadata tableMetadata)`：通过查询的结果集创建表，也就是 `Create Table As Select` 的操作，分成两个阶段完成，`beginCreateTable` 为第一个阶段，`commitCreateTable` 为第二个阶段。`beginCreateTable` 这部分主要是获取建表时所需的元数据信息，例如字段信息等，然后将这些信息封装成 `ConnectorOutputTableHandle` 对象，提供给下一阶段使用。
- `public void commitCreateTable(ConnectorOutputTableHandle tableHandle, Collection<Slice> fragments)`：完成了 `beginCreateTable` 之后的操作内容，`commitCreateTable` 负责将表和数据相对应以及提交创建表信息等操作。

- `public void renameTable(ConnectorTableHandle tableHandle, SchemaTableName newTableName)`：重命名表。
- `public ConnectorInsertTableHandle beginInsert(ConnectorSession session, ConnectorTableHandle tableHandle, InsertOption insertOption)`：对应的是 `insert into table` 的操作，与 `create table as Select` 操作类似，`insert` 操作也分成了两个阶段，`beginInsert` 为第一个阶段，该方获取 `insert` 操作所需的表元数据信息等，例如 Hive Connector 中的该方法将完成获取表的元数据、创建临时的数据目录地址等，并将这些信息封装为 `ConnectorInsertTableHandle` 对象，提供给下一阶段使用。
- `public void commitInsert(ConnectorInsertTableHandle tableHandle, Collection<Slice> fragments)`：执行第二阶段的 `insert` 操作，该阶段完成数据与表的对应关系以及相关元数据的创建，比如 Hive Connector 中的该方法将临时目录的数据最终移到表所在的数据目录，如果存在创建分区操作，创建分区的操作也在该阶段完成。
- `public void createView(ConnectorSession session, SchemaTableName viewName, String viewData, boolean replace)`：创建 view 视图。
- `public void dropView(ConnectorSession session, SchemaTableName viewName)`：删除 view 视图。

12.5 SplitManager

Presto 中的 Split 详细描述了一个数据分片的具体内容，每种数据源的 Split 都不尽相同，数据源中定义的 Split 即为 Source Stage 中调度的 Split。例如 hive Connector 定义的 Split 描述的是 hdfs 文件的相关信息，而 mysql Connector 定义的 Split 则是描述的连接 MySQL 数据库表的相关信息，而且 Split 的分片合理性直接决定了 Presto 读取数据源数据的效率。如果一个 Split 分片涵盖的数据量过大，数据不均匀，则会拖慢整个 SQL；若过小，则会造成大量的 CPU 资源都耗费在调度 Split 的操作上。Hive Connector 通过配置参数 `hive.max-split-size` 定义一个 Split 的最大数据量 `maxSize`，默认为 64MB，尽量和 HDFS 的 block size 的大小保持一致。

Presto source stage splits 通过 `com.facebook.presto.split.SplitManager` 获取，最终是通过接口 `com.facebook.presto.spi.ConnectorSplitManager` 的实现类，也就是各个 Connector 的 `SplitManager` 获取 Split，该接口包括了以下两个方法：

- `ConnectorPartitionResult getPartitions(ConnectorTableHandle table, TupleDomain<ConnectorColumnHandle> tupleDomain)`：获取表的分区信息。
- `ConnectorSplitSource getPartitionSplits(ConnectorTableHandle table, List<ConnectorPartition> partitions)`：获取表的 Splits。

其中 `ConnectorSplitSource` 为一个 Connector 中具体表的 Split 集合，主要包含以下几个方法。

- `String getDataSourceName()`：返回的是数据源名称。

- `List<ConnectorSplit> getNextBatch(int maxSize)`: 批量获取 Splits 列表, `maxSize` 默认为 1000, Split 如何组织以及每调用一次该方法如何返回 `List<ConnectorSplit>` 集合都由各个 Connector 自行定义。

除了 Hive Connector 和 Raptor Connector 有单独的 `ConnectorSplitSource` 实现类, 其余的 Connector 均使用了 `com.facebook.presto.spi.FixedSplitSource` 作为其实现类。`FixedSplitSource` 的实现也是非常简单, 代码如下:

```
public FixedSplitSource(String dataSourceName, Iterable<? extends
ConnectorSplit> splits)
{
    this.dataSourceName = dataSourceName;
    if (splits == null) {
        throw new NullPointerException("splits is null");
    }
    // 重新复制一份 split 列表
    List<ConnectorSplit> splitsList = new ArrayList<>();
    for (ConnectorSplit split : splits) {
        splitsList.add(split);
    }
    // 针对 splitsList 生成一个只读视图
    this.splits = Collections.unmodifiableList(splitsList);
}
@Override
public String getDataSourceName()
{
    return dataSourceName;
}
@Override
public List<ConnectorSplit> getNextBatch(int maxSize)
    throws InterruptedException
{
    int remainingSplits = splits.size() - offset;
    int size = Math.min(remainingSplits, maxSize);
    // 从前往后顺序获取
    List<ConnectorSplit> results = splits.subList(offset, offset + size);
    offset += size;
    return results;
}
@Override
public boolean isFinished()
{
    return offset >= splits.size();
}
}
```

12.6 RecordSetProvider

Presto 获取到 Source Split 后, 将其调度到具体的 Work 节点上执行数据读取, 数据读取过程需要依赖 Split 对应的 Connector 所提供的的数据读取类, 也就是 `RecordSetProvider`。

数据源的 `RecordSetProvider` 均需要实现 `com.facebook.presto.spi.ConnectorRecordSetProvider` 接口，接口定义也很简单，提供了输入参数是 `Split` 和表的列信息及输出为 `RecordSet` 的方法，代码如下：

```
public interface ConnectorRecordSetProvider
{
    RecordSet getRecordSet(ConnectorSplit split, List<? extends
ConnectorColumnHandle> columns);
}
```

其中，`RecordSet` 为 `com.facebook.presto.spi.RecordSet`，其定义的方法有：

```
public interface RecordSet
{
    List<Type> getColumnTypes();
    RecordCursor cursor();
}
```

`cursor` 方法返回的 `com.facebook.presto.spi.RecordCursor` 就是真正读取数据类，其定义了 `Split` 的数据读取方法，针对每个 `Split` 都会创建一个 `RecordCursor` 实例，该实例提供以下方法。

- `long getTotalBytes()`：获取 `Split` 的总字节数。
- `long getCompletedBytes()`：获取 `Split` 已经读取完成的字节总数。
- `long getReadTimeNanos()`：获取数据读取时长。
- `Type getType(int field)`：获取指定字段的数据类型。
- `boolean advanceNextPosition()`：判断当前 `Split` 是否还有下一行数据，也就是判断是否还有数据未读取完成。
- `boolean getBoolean(int field)`：获取 `boolean` 型字段的值。
- `long getLong(int field)`：获取整型字段的值。
- `double getDouble(int field)`：获取 `double` 型字段的值。
- `Slice getSlice(int field)`：获取字符串型字段的值。
- `boolean isNull(int field)`：判断当前字段值是否为 `null`。
- `void close()`：关闭当前读取进程操作。

12.7 小结

本章以 `presto-example-http` 为例，详细讲解了自定义 `Connector` 的开发步骤和需要注意的问题，并且讲述了 `Connector` 中各个组件所涵盖的功能以及实现过程。不过本章所讲述的 `presto-example-http` `Connector` 中的各个组件实现相对简单，仅仅实现了大体框架，对于功能和性能要求更加复杂的 `Connector`，例如 `Hive Connector`，需要实现的细节以及扩展比本章所讲述的内容要更复杂。

第 13 章

Functions 开发

Presto 的 Functions 是使用在 SQL 语句中，用于实现特定功能的函数。Presto 内部实现了一系列的 Functions，提供了非常丰富的 Function 操作，常用的 Functions 包括以下几类。

- 比较类型函数，例如：greatest, least。
- 条件类型函数，例如：case, if。
- 类型转换函数，例如：cast, type_cast。
- 数学函数，例如：abs, ceil, floor。
- 字符相关操作函数，例如：concat, length, replace。
- 时间函数，例如：current_date, now, date_add。
- 正则函数，例如：regexp_like, regexp_replace。
- JSON 相关函数，例如：json_array_length, json_extract。
- URL 相关函数，例如：url_extract_fragment, url_extract_host。
- 聚合函数，例如：sum, count。
- 窗口函数，例如：row_number, rank。
- 复杂类型操作函数，例如：Array/Map。

详细说明可以参考官方网站提供的使用手册：<https://presto.io/docs/current/functions.html>，本章会从函数注册到函数的具体实现详细讲解 Presto Functions 的开发。

13.1 Function 注册

Function 的注册是通过 `com.facebook.presto.metadata.FunctionRegistry` 来完成的，在 `com.facebook.presto.metadata.MetadataManager` 构造方法中会初始化 `FunctionRegistry` 实例，在 `FunctionRegistry` 的构造方法中，注册了目前 Presto 中所有的 Functions：

```
public FunctionRegistry(TypeManager typeManager, BlockEncodingSerde
blockEncodingSerde, boolean experimentalSyntaxEnabled)
```

```

{
    .....

    FunctionListBuilder builder = new FunctionListBuilder(typeManager)
        .window("row_number", BIGINT, ImmutableList.<Type>of(),
RowNumberFunction.class)
        .window("rank", BIGINT, ImmutableList.<Type>of(),
RankFunction.class)
        .window("dense_rank", BIGINT, ImmutableList.<Type>of(),
DenseRankFunction.class)
        .window("percent_rank", DOUBLE, ImmutableList.<Type>of(),
PercentRankFunction.class)
        .window("cume_dist", DOUBLE, ImmutableList.<Type>of(),
CumulativeDistributionFunction.class)
        .window("ntile", BIGINT, ImmutableList.<Type>of(BIGINT),
NTileFunction.class)
        .window("first_value", BIGINT, ImmutableList.<Type>of(BIGINT),
BigintFirstValueFunction.class)
        .window("first_value", DOUBLE, ImmutableList.<Type>of(DOUBLE),
DoubleFirstValueFunction.class)
        .window("first_value", BOOLEAN, ImmutableList.<Type>of(BOOLEAN),
BooleanFirstValueFunction.class)
        .window("first_value", VARCHAR, ImmutableList.<Type>of(VARCHAR),
VarcharFirstValueFunction.class)
        .window("first_value", TIMESTAMP,
ImmutableList.<Type>of(TIMESTAMP), TimestampFirstValueFunction.class)
        .window("last_value", BIGINT, ImmutableList.<Type>of(BIGINT),
BigintLastValueFunction.class)
        .window("last_value", DOUBLE, ImmutableList.<Type>of(DOUBLE),
DoubleLastValueFunction.class)
        .window("last_value", BOOLEAN, ImmutableList.<Type>of(BOOLEAN),
BooleanLastValueFunction.class)
        .window("last_value", VARCHAR, ImmutableList.<Type>of(VARCHAR),
VarcharLastValueFunction.class)
        .window("last_value", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP),
TimestampLastValueFunction.class)
        .window("nth_value", BIGINT, ImmutableList.<Type>of(BIGINT,
BIGINT), BigintNthValueFunction.class)
        .window("nth_value", DOUBLE, ImmutableList.<Type>of(DOUBLE,
BIGINT), DoubleNthValueFunction.class)
        .window("nth_value", BOOLEAN, ImmutableList.<Type>of(BOOLEAN,
BIGINT), BooleanNthValueFunction.class)
        .window("nth_value", VARCHAR, ImmutableList.<Type>of(VARCHAR,
BIGINT), VarcharNthValueFunction.class)
        .window("nth_value", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP,
BIGINT), TimestampNthValueFunction.class)
        .window("lag", BIGINT, ImmutableList.<Type>of(BIGINT),
BigintLagFunction.class)
        .window("lag", BIGINT, ImmutableList.<Type>of(BIGINT, BIGINT),
BigintLagFunction.class)
        .window("lag", BIGINT, ImmutableList.<Type>of(BIGINT, BIGINT,
BIGINT), BigintLagFunction.class)
        .window("lag", DOUBLE, ImmutableList.<Type>of(DOUBLE),

```



```

DoubleLagFunction.class)
    .window("lag", DOUBLE, ImmutableList.<Type>of(DOUBLE, BIGINT),
DoubleLagFunction.class)
    .window("lag", DOUBLE, ImmutableList.<Type>of(DOUBLE, BIGINT,
DOUBLE), DoubleLagFunction.class)
    .window("lag", BOOLEAN, ImmutableList.<Type>of(BOOLEAN),
BooleanLagFunction.class)
    .window("lag", BOOLEAN, ImmutableList.<Type>of(BOOLEAN, BIGINT),
BooleanLagFunction.class)
    .window("lag", BOOLEAN, ImmutableList.<Type>of(BOOLEAN, BIGINT,
BOOLEAN), BooleanLagFunction.class)
    .window("lag", VARCHAR, ImmutableList.<Type>of(VARCHAR),
VarcharLagFunction.class)
    .window("lag", VARCHAR, ImmutableList.<Type>of(VARCHAR, BIGINT),
VarcharLagFunction.class)
    .window("lag", VARCHAR, ImmutableList.<Type>of(VARCHAR, BIGINT,
VARCHAR), VarcharLagFunction.class)
    .window("lag", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP),
TimestampLagFunction.class)
    .window("lag", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP,
BIGINT), TimestampLagFunction.class)
    .window("lag", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP, BIGINT,
VARCHAR), TimestampLagFunction.class)
    .window("lead", BIGINT, ImmutableList.<Type>of(BIGINT),
BigintLeadFunction.class)
    .window("lead", BIGINT, ImmutableList.<Type>of(BIGINT, BIGINT),
BigintLeadFunction.class)
    .window("lead", BIGINT, ImmutableList.<Type>of(BIGINT, BIGINT,
BIGINT), BigintLeadFunction.class)
    .window("lead", DOUBLE, ImmutableList.<Type>of(DOUBLE),
DoubleLeadFunction.class)
    .window("lead", DOUBLE, ImmutableList.<Type>of(DOUBLE, BIGINT),
DoubleLeadFunction.class)
    .window("lead", DOUBLE, ImmutableList.<Type>of(DOUBLE, BIGINT,
DOUBLE), DoubleLeadFunction.class)
    .window("lead", BOOLEAN, ImmutableList.<Type>of(BOOLEAN),
BooleanLeadFunction.class)
    .window("lead", BOOLEAN, ImmutableList.<Type>of(BOOLEAN, BIGINT),
BooleanLeadFunction.class)
    .window("lead", BOOLEAN, ImmutableList.<Type>of(BOOLEAN, BIGINT,
BOOLEAN), BooleanLeadFunction.class)
    .window("lead", VARCHAR, ImmutableList.<Type>of(VARCHAR),
VarcharLeadFunction.class)
    .window("lead", VARCHAR, ImmutableList.<Type>of(VARCHAR, BIGINT),
VarcharLeadFunction.class)
    .window("lead", VARCHAR, ImmutableList.<Type>of(VARCHAR, BIGINT,
VARCHAR), VarcharLeadFunction.class)
    .window("lead", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP),
TimestampLeadFunction.class)
    .window("lead", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP,
BIGINT), TimestampLeadFunction.class)
    .window("lead", TIMESTAMP, ImmutableList.<Type>of(TIMESTAMP,

```



```

BIGINT, VARCHAR), TimestampLeadFunction.class)
    .aggregate(CountAggregation.class)
    .aggregate(VarianceAggregation.class)
    .aggregate(ApproximateLongPercentileAggregations.class)
    .aggregate(ApproximateDoublePercentileAggregations.class)
    .aggregate(CountIfAggregation.class)
    .aggregate(BooleanAndAggregation.class)
    .aggregate(BooleanOrAggregation.class)
    .aggregate(DoubleSumAggregation.class)
    .aggregate(LongSumAggregation.class)
    .aggregate(AverageAggregations.class)
    .aggregate(ApproximateCountDistinctAggregations.class)
    .aggregate(MergeHyperLogLogAggregation.class)
    .aggregate(ApproximateSetAggregation.class)
    .aggregate(NumericHistogramAggregation.class)
    .aggregate(CovarianceAggregation.class)
    .aggregate(RegressionAggregation.class)
    .aggregate(CorrelationAggregation.class)
    .scalar(StringFunctions.class)
    .scalar(VarbinaryFunctions.class)
    .scalar(RegexpFunctions.class)
    .scalar(UrlFunctions.class)
    .scalar(MathFunctions.class)
    .scalar(DateTimeFunctions.class)
    .scalar(JsonFunctions.class)
    .scalar(ColorFunctions.class)
    .scalar(HyperLogLogFunctions.class)
    .scalar(BooleanOperators.class)
    .scalar(BigintOperators.class)
    .scalar(DoubleOperators.class)
    .scalar(VarcharOperators.class)
    .scalar(VarbinaryOperators.class)
    .scalar(DateOperators.class)
    .scalar(TimeOperators.class)
    .scalar(TimestampOperators.class)
    .scalar(IntervalDayTimeOperators.class)
    .scalar(IntervalYearMonthOperators.class)
    .scalar(TimeWithTimeZoneOperators.class)
    .scalar(TimestampWithTimeZoneOperators.class)
    .scalar(DateTimeOperators.class)
    .scalar(HyperLogLogOperators.class)
    .scalar(LikeFunctions.class)
    .scalar(ArrayFunctions.class)
    .scalar(CombineHashFunction.class)
    .scalar(JsonOperators.class)
    .function(IDENTITY_CAST)
    .functions(ARRAY_CONTAINS, ARRAY_JOIN, ARRAY_JOIN_WITH_NULL_
REPLACEMENT)
    .functions(ARRAY_TO_ARRAY_CAST, ARRAY_HASH_CODE, ARRAY_EQUAL,
ARRAY_NOT_EQUAL, ARRAY_LESS_THAN, ARRAY_LESS_THAN_OR_EQUAL, ARRAY_GREATER_THAN,
ARRAY_GREATER_THAN_OR_EQUAL)
    .functions(ARRAY_CONCAT_FUNCTION, ARRAY_TO_ELEMENT_CONCAT

```

```

FUNCTION, ELEMENT_TO_ARRAY_CONCAT_FUNCTION)
        .functions(MAP_EQUAL, MAP_NOT_EQUAL, MAP_HASH_CODE)
        .functions(ARRAY_CONSTRUCTOR, ARRAY_SUBSCRIPT, ARRAY_CARDINALITY,
ARRAY_POSITION, ARRAY_SORT_FUNCTION, ARRAY_INTERSECT_FUNCTION, ARRAY_TO_JSON,
JSON_TO_ARRAY, ARRAY_DISTINCT_FUNCTION, ARRAY_REMOVE_FUNCTION)
        .functions(MAP_CONSTRUCTOR, MAP_CARDINALITY, MAP_SUBSCRIPT,
MAP_TO_JSON, JSON_TO_MAP, MAP_KEYS, MAP_VALUES, MAP_AGG)
        .function(ARBITRARY_AGGREGATION)
        .function(ARRAY_AGGREGATION)
        .function(LEAST)
        .function(GREATEST)
        .function(MAX_BY)
        .function(MIN_BY)
        .functions(MAX_AGGREGATION, MIN_AGGREGATION)
        .function(COUNT_COLUMN)
        .functions(ROW_HASH_CODE, ROW_TO_JSON, ROW_EQUAL, ROW_NOT_EQUAL)
        .function(TRY_CAST);
.....
addFunctions(builder.getFunctions());

```

最终通过 `addFunctions` 方法，将 `FunctionListBuilder` 中构造的 `Function` 加入到 `FunctionMap`，`FunctionMap` 维护了一个 `function name` 为 `key`，`function information` 为 `value` 的 `map`。需要注意的是，对同一个类型字段的 `function name` 不能重复，否则将会抛异常。

在注册过程中，Presto 会将各个类型的 `Function` 分类，例如 `row_number` 为窗口函数，`string function` 中注册了操作字符串的各种扩展函数，`sum`、`count` 等操作被注册为聚合函数，还有一些操作复杂类型的函数，例如 `json`、`array`、`map` 等。

除在 `FunctionRegistry` 的构造方法中所注册的 `Function` 外，在 `com.facebook.presto.server.PluginManager` 插件管理中加载 `Plugin` 时，也可以在各个 `Plugin` 中向 `FunctionRegistry` 注册自定义 `Function`，如以下代码所示：

```

public void installPlugin(Plugin plugin)
{
    for (FunctionFactory functionFactory :
plugin.getServices(FunctionFactory.class)) {
        log.info("Registering functions from %s", functionFactory.getClass().
getName());
        metadata.addFunctions(functionFactory.listFunctions());
    }
}

```

也就是在各个 `Plugin` 的 `getServices` 方法中，可以注册自定义的 `Functions`。

Presto 在 `cli` 命令行或者通过 `Presto jdbc` 连接执行 `show functions` 命令，即可返回所有当前 Presto 注册成功的 `Functions`。在源码层面，`show functions` 命令最终会返回的是上述讲到的 `FunctionMap` 维护的 `map` 的 `values` 集合，如图 13-1 所示。

Function	Return Type	Argument Types	Function Type	Deterministic	Description
abs	bigint	bigint	scalar	true	absolute value
abs	double	double	scalar	true	absolute value
acos	double	double	scalar	true	arc cosine
approx_distinct	bigint	bigint	aggregate	true	
approx_distinct	bigint	bigint, double	aggregate	true	
approx_distinct	bigint	double	aggregate	true	
approx_distinct	bigint	double, double	aggregate	true	
approx_distinct	bigint	varchar	aggregate	true	
approx_distinct	bigint	varchar, double	aggregate	true	
approx_percentile	double	bigint, bigint, double	aggregate	true	
approx_percentile	double	bigint, double	aggregate	true	
approx_percentile	double	double, bigint, double	aggregate	true	
approx_percentile	double	double, double	aggregate	true	
approx_set	hyperloglog	bigint	aggregate	true	
approx_set	hyperloglog	double	aggregate	true	
approx_set	hyperloglog	varchar	aggregate	true	
array_distinct	array<T>	array<T>	scalar	true	return an arbitrary non-null input value
array_sort	array<T>	array<T>	scalar	true	Remove duplicate values from the given array
asin	double	double	scalar	true	Sorts the given array in ascending order according to the natural arc sine
at_timezone	time with time zone	time with time zone, interval day to second	scalar	true	
at_timezone	time with time zone	time with time zone, varchar	scalar	true	
at_timezone	timestamp with time zone	timestamp with time zone, interval day to second	scalar	true	
at_timezone	timestamp with time zone	timestamp with time zone, varchar	scalar	true	
atan	double	double	scalar	true	arc tangent
atan2	double	double, double	scalar	true	arc tangent of given fraction
avg	double	double	aggregate	true	
avg	double	double, bigint	aggregate	true	
bar	varchar	double, bigint, color, color	scalar	true	
bar	varchar	double, bigint, color, color	scalar	true	
bool_and	boolean	boolean	aggregate	true	
bool_or	boolean	boolean	aggregate	true	
cardinality	bigint	hyperloglog	scalar	false	compute the cardinality of a hyperloglog instance
cardinality	bigint	array<T>	scalar	false	
cardinality	bigint	map<K, V>	scalar	false	same as set

图 13-1

13.2 窗口函数

窗口函数可以被理解和分析型函数，即将某一系列多行中的值按照一定的聚合规则进行计算，例如我们熟知的 SUM、AVG 函数等。Presto 默认提供了很多的窗口函数，详细可以查看：<https://presto.io/docs/current/functions/window.html>。Presto 中的窗口函数默认要和 over 函数配合使用，over 函数中可以定义分组和排序规则，例如常用的 row_number 函数，如图 13-2 所示。

```
presto:employees> select emp_no, salary, row_number() over(partition by emp_no order by emp_no) from salaries limit 100;
emp_no | salary | col2
-----|-----|-----
10009  | 60929  | 1
10009  | 94409  | 2
10009  | 69042  | 3
10009  | 64604  | 4
10009  | 64780  | 5
10009  | 70889  | 6
10009  | 71434  | 7
10009  | 74612  | 8
10009  | 76518  | 9
10009  | 78335  | 10
10009  | 80944  | 11
10009  | 82507  | 12
10009  | 85875  | 13
10009  | 89324  | 14
10009  | 90668  | 15
10009  | 94443  | 16
10009  | 93507  | 17
10009  | 66302  | 18
11465  | 40000  | 1
11465  | 71537  | 2
11465  | 51646  | 3
11465  | 43472  | 4
11465  | 44906  | 5
11465  | 55274  | 6
11465  | 59313  | 7
11465  | 60710  | 8
11465  | 65157  | 9
```

图 13-2

在图 13-2 中对 emp_no 分组并排序，输出 emp_no、salary、row_number。以下将开发一个简单的窗口函数样例来展示整个窗口函数开发的流程。

窗口函数需要继承 com.facebook.presto.operator.window.WindowFunction 接口，WindowFunction 定义了三个方法：

- `Type getType()`: 获取当前列的数据类型。
- `void reset(WindowIndex windowIndex)`: 初始化窗口函数时将会调用该方法, `WindowIndex` 包含了返回的结果集信息。
- `void processRow(BlockBuilder output, int peerGroupStart, int peerGroupEnd, int frameStart, int frameEnd)`: 窗口函数功能的实现方法, `BlockBuilder` 为结果输出类, 计算出来的值将通过 `BlockBuilder` 进行输出; `int peerGroupStart` 和 `int peerGroupEnd` 根据某列分组后, 每组的开始和结束位置; `int frameStart` 和 `int frameEnd` 为结果集的开始位置和结束位置。

接下来将示例新增一个新的窗口函数, 此窗口函数的功能是: 对某列的值进行逐一累计进行合计, 以 `employees` 表为实验数据, 该表的部分数据如图 13-3 所示。

emp_no	salary	from_date	to_date
10001	60117	1986-06-26	1987-06-26
10001	62102	1987-06-26	1988-06-25
10001	66074	1988-06-25	1989-06-25
10001	66596	1989-06-25	1990-06-25
10001	66961	1990-06-25	1991-06-25
10001	71046	1991-06-25	1992-06-24
10001	74333	1992-06-24	1993-06-24
10001	75286	1993-06-24	1994-06-24
10001	75994	1994-06-24	1995-06-24
10001	76884	1995-06-24	1996-06-23
10001	80013	1996-06-23	1997-06-23
10001	81025	1997-06-23	1998-06-23
10001	81097	1998-06-23	1999-06-23
10001	84917	1999-06-23	2000-06-22
10001	85112	2000-06-22	2001-06-22
10001	85097	2001-06-22	2002-06-22
10001	88958	2002-06-22	9999-01-01
10002	65828	1996-08-03	1997-08-03
10002	65909	1997-08-03	1998-08-03
10002	67534	1998-08-03	1999-08-03
10002	69366	1999-08-03	2000-08-02
10002	71963	2000-08-02	2001-08-02
10002	72527	2001-08-02	9999-01-01

(23 rows)

图 13-3

图 13-3 显示的结果集为 `emp_no` 为 10001 和 10002 每年的工资信息, 需要统计出员工每年工资逐一累加的小计值, 效果如图 13-4 所示。

emp_no	salary	from_date	to_date	salary_sum
10001	60117	1986-06-26	1987-06-26	60117
10001	62102	1987-06-26	1988-06-25	122219
10001	66074	1988-06-25	1989-06-25	188293
10001	66596	1989-06-25	1990-06-25	254889
10001	66961	1990-06-25	1991-06-25	321850
10001	71046	1991-06-25	1992-06-24	392896
10001	74333	1992-06-24	1993-06-24	467229
10001	75286	1993-06-24	1994-06-24	542515
10001	75994	1994-06-24	1995-06-24	618509
10001	76884	1995-06-24	1996-06-23	695393
10001	80013	1996-06-23	1997-06-23	775406
10001	81025	1997-06-23	1998-06-23	856431
10001	81097	1998-06-23	1999-06-23	937528
10001	84917	1999-06-23	2000-06-22	1022445
10001	85112	2000-06-22	2001-06-22	1107557
10001	85097	2001-06-22	2002-06-22	1192654
10001	88958	2002-06-22	9999-01-01	1281612
10002	65828	1996-08-03	1997-08-03	65828
10002	65909	1997-08-03	1998-08-03	131737
10002	67534	1998-08-03	1999-08-03	199271
10002	69366	1999-08-03	2000-08-02	268637
10002	71963	2000-08-02	2001-08-02	340600
10002	72527	2001-08-02	9999-01-01	413127

(23 rows)

图 13-4

首先将这个新增的窗口函数命名为 `sum_by`，为此首先在 `presto-main module` 中的 `com.facebook.presto.operator.window` 目录下新增一个新的 `function class`：`com.facebook.presto.operator.window.SumByFunction`，然后限定使用范围是针对整型和浮点型的字段，代码如下：

```
public class SumByFunction implements WindowFunction
{
    // 定义操作 double 类型的 function 实现类
    public static class DoubleSumByFunction extends SumByFunction
    {
        public DoubleSumByFunction(List<Double> argumentChannels)
        {
            super(DOUBLE, argumentChannels);
        }
    }
    // 定义操作 int/bigint 类型的 function 实现类
    public static class BigintSumByFunction extends SumByFunction
    {
        public BigintSumByFunction(List<Integer> argumentChannels)
        {
            super(BIGINT, argumentChannels);
        }
    }
}
```

在 Presto 中整型和浮点型都用 `bigint` 和 `double` 处理。

然后依次实现 `WindowFunction` 接口的三个方法：

```
private final Type type;
private final int argumentChannel;
private WindowIndex windowIndex;
private int currentPeerGroupStart;
private int currentPosition;
private long sumLongValue;
private double sumDoubleValue;
protected SumByFunction(Type type, List<Integer> argumentChannels)
{
    this.type = type;
    this.argumentChannel = getOnlyElement(argumentChannels);
}
@Override
public Type getType()
{
    // 获取字段类型
    return type;
}
@Override
public void reset(WindowIndex windowIndex)
{
    // 初始相关变量的方法
    this.windowIndex = windowIndex;
    this.currentPeerGroupStart = -1;
    this.currentPosition = 0;
    this.sumLongValue = 0L;
    this.sumDoubleValue = 0d;
}
@Override
public void processRow(BlockBuilder output, int peerGroupStart, int peerGroupEnd,
int frameStart, int frameEnd)
{
    boolean newPeerGroup = false;
    if (peerGroupStart != currentPeerGroupStart) {
        currentPeerGroupStart = peerGroupStart;
        newPeerGroup = true;
    }
    // 是否为新的分组
    if (newPeerGroup) {
        this.sumLongValue = 0L;
        this.sumDoubleValue = 0d;
    }
    if (type instanceof BigintType) {
        long curValue = windowIndex.getLong(argumentChannel, currentPosition);
        sumLongValue += curValue;
        BIGINT.writeLong(output, sumLongValue);
    }
    else {
        double doubleValue = windowIndex.getDouble(argumentChannel,
```

```

currentPosition);
    sumDoubleValue += doubleValue;
    DOUBLE.writeDouble(output, sumDoubleValue);
}
currentPosition++;
}

```

其中 `argumentChannel` 为当前列的索引值，`currentPosition` 为当前行的索引值，所以可以由 `argumentChannel` 和 `currentPosition` 来确定某个字段某行的值。`newPeerGroup` 用来确定当前行是否为新的分组，例如图 13-4 中的效果图所示就是对 `emp_no` 进行了分组，若不进行分组，`salary_sum` 列将一直累加。

窗口函数实现完成后，在 `FunctionRegistry` 中进行注册，注册代码如下：

```

public FunctionRegistry(TypeManager typeManager, boolean
experimentalSyntaxEnabled)
{
    .....

    FunctionListBuilder builder = new FunctionListBuilder(typeManager)
        .....
        .window("sum_by", BIGINT, ImmutableList.<Type>of(BIGINT), SumByFunction.
BigintSumByFunction.class)
        .window("sum_by", DOUBLE, ImmutableList.<Type>of(DOUBLE), SumByFunction.
DoubleSumByFunction.class)
        .....
}

```

这样便完成了注册，重新编译 `presto-main module`，然后替换 `$PRESTO_HOME/lib/presto-main-$VERSION.jar`，重启 Presto 集群即可使用。图 13-4 产生的结果来自于执行以下 SQL 语句得到的：

```

select emp_no,salary,from_date,to_date,sum_by(salary) over(partition by emp_no order by
emp_no) salary_sum from salaries where emp_no in ('10001','10002');

```

13.3 聚合函数

聚合函数的实现是用注解框架来实现的，聚合函数分别需要定义函数名称，函数的输入、输出以及聚合过程。

@AggregationFunction：单元素注解。声明聚合函数名称，以及标注当前函数为聚合函数。

@InputFunction：标记注解。注解的方法接受输入参数，并对输入参数进行处理。

@OutputFunction：单元素注解。声明输出结果的类型。注解的方法完成最终结果的输出结果输出。

@CombineFunction：标记注解。注解的方法完成两个数据的聚合。

例如以下例子为实现 `avg` 方法的代码：


```

@AggregationFunction("avg")
public final class AverageAggregations
{
    @InputFunction
    public static void input(LongAndDoubleState state, @SqlType(StandardTypes.
BIGINT) long value)
    {
        // 记录计算平均值所需的内容:
        // 总记录值以及每行内容的总和
        state.setLong(state.getLong() + 1);
        state.setDouble(state.getDouble() + value);
    }

    @InputFunction
    public static void input(LongAndDoubleState state, @SqlType(StandardTypes.
DOUBLE) double value)
    {
        // 记录计算平均值所需的内容:
        // 总记录值以及每行内容的总和
        state.setLong(state.getLong() + 1);
        state.setDouble(state.getDouble() + value);
    }

    @CombineFunction
    public static void combine(LongAndDoubleState state, LongAndDoubleState
otherState)
    {
        // 合并两个记录了总记录值以及每行内容的值
        state.setLong(state.getLong() + otherState.getLong());
        state.setDouble(state.getDouble() + otherState.getDouble());
    }

    @OutputFunction(StandardTypes.DOUBLE)
    public static void output(LongAndDoubleState state, BlockBuilder out)
    {
        long count = state.getLong();
        if (count == 0) {
            out.appendNull();
        }
        else {
            double value = state.getDouble();
            // 最终的结果输出
            DOUBLE.writeDouble(out, value / count);
        }
    }
}

```

聚合函数的注册过程也正是根据注解来完成的，聚合函数的注册主要通过 `FunctionListBuilder` 中的 `aggregate` 方法来完成，代码如下：

```

public FunctionListBuilder aggregate(Class<?> aggregationDefinition)
{
    functions.addAll(GenericAggregationFunctionFactory.fromAggregationDefinition
(aggregationDefinition, typeManager).listFunctions());
    return this;
}

```


GenericAggregationFunctionFactory.fromAggregationDefinition 的源码如下：

```

    }

    public static GenericAggregationFunctionFactory fromAggregationDefinition
    (Class<?> clazz, TypeManager typeManager)
    {
        FunctionListBuilder builder = new FunctionListBuilder(typeManager);
        for (InternalAggregationFunction aggregation : new AggregationCompiler
        (typeManager).generateAggregationFunctions(clazz)) {
            builder.aggregate(aggregation);
        }

        return new GenericAggregationFunctionFactory(builder.getFunctions());
    }

```

最终通过 AggregationCompiler.generateAggregationFunctions 方法来完成 Function 解析和注册，该方法的源码如下：

```

    public List<InternalAggregationFunction> generateAggregationFunctions(Class<?>
    clazz)
    {
        AggregationFunction aggregationAnnotation = clazz.getAnnotation
        (AggregationFunction.class);
        checkNotNull(aggregationAnnotation, "aggregationAnnotation is null");

        DynamicClassLoader classLoader = new
        DynamicClassLoader(clazz.getClassLoader());

        ImmutableList.Builder<InternalAggregationFunction> builder =
        ImmutableList.builder();
        for (Class<?> stateClass : getStateClasses(clazz)) {
            AccumulatorStateSerializer<?> stateSerializer = new
            StateCompiler().generateStateSerializer(stateClass, classLoader);
            Type intermediateType = stateSerializer.getSerializedType();
            Method intermediateInputFunction = getIntermediateInputFunction(clazz,
            stateClass);
            Method combineFunction = getCombineFunction(clazz, stateClass);
            AccumulatorStateFactory<?> stateFactory = new
            StateCompiler().generateStateFactory(stateClass, classLoader);

            for (Method outputFunction : getOutputFunctions(clazz, stateClass)) {
                for (Method inputFunction : getInputFunctions(clazz, stateClass)) {
                    for (String name : getNames(outputFunction, aggregationAnnotation))
                    {
                        List<Type> inputTypes = getInputTypes(inputFunction);
                        Type outputType = AggregationUtils.getOutputType(outputFunction,
                        stateSerializer, typeManager);

                        AggregationMetadata metadata = null;
                        try {
                            MethodHandle inputHandle =
                            lookup().unreflect(inputFunction);

```

```

        MethodHandle intermediateInputHandle =
intermediateInputFunction == null ? null :
lookup().unreflect(intermediateInputFunction);
        MethodHandle combineHandle = combineFunction == null ? null :
lookup().unreflect(combineFunction);
        MethodHandle outputHandle = outputFunction == null ? null :
lookup().unreflect(outputFunction);
        metadata = new AggregationMetadata(
            generateAggregationName(name, outputType,
inputTypes),
            getParameterMetadata(inputFunction,
aggregationAnnotation.approximate()),
            inputHandle,
            getParameterMetadata(intermediateInputFunction,
false),
            intermediateInputHandle,
            combineHandle,
            outputHandle,
            stateClass,
            stateSerializer,
            stateFactory,
            outputType,
            aggregationAnnotation.approximate());
    }
    catch (IllegalAccessException e) {
        throw Throwables.propagate(e);
    }
    GenericAccumulatorFactoryBinder factory = new
AccumulatorCompiler().generateAccumulatorFactoryBinder(metadata, classLoader);
    builder.add(new InternalAggregationFunction(name, inputTypes,
intermediateType, outputType, aggregationAnnotation.decomposable(),
aggregationAnnotation.approximate(), factory));
    }
    }
    }
    return builder.build();
}

```

13.4 小结

Presto Functions 并不能像 Hive UDF 一样动态加载，需要根据 Function 的具体功能，实现 Presto 内部定义的各个接口。比较常用的 Functions 主要包括：窗口函数和聚合函数，当然也会对 Presto 自带的函数进行扩展，例如针对字符串各种操作的扩展等，通过本章，读者可以深入了解 Presto 中 Function 的定义、注册的流程和原理，并且能够根据自己的需要开发自定义 Function。

第三部分

高级篇

第 14 章

JD-Presto 功能改造

Presto 在大数据分布式计算方面表现出了非常良好的性能，数据源的扩展提供了非常友好的接口。京东 (jd.com) 为了满足自身的业务需求，对某些功能进行了扩展，比如 PDBO 功能模块是为了让 Presto 更好地支持关系型数据库的数据源；同时为了使原有的一些业务迁移或者是使用方式的延续，JD-Presto 版本针对 Hive-Connector 使其支持了 DDL 和 DML 操作，增加了使用的灵活性；在 Presto 集群维护层面上，为了减少集群重启以及频繁地修改 Catalog 配置文件，也特地增加了可动态增删改 Catalog 配置文件的功能。

除以上新增的功能外，京东 Presto 团队还积极参与官方 Presto 开源社区建设，向 Presto 社区提交 code，在国内积极推广 Presto，并建立了 Presto 中文社区：<http://prestodb-china.com/>。本章会详细介绍 JD-Presto 版本中几个新增的较为重要的功能模块。

14.1 PDBO 功能开发

直至 Presto 最新版本，Presto 处理 MySQL 数据的模式为：每个 Source Stage 只有一个 Presto Work 处理一个 Split。换言之。当使用 Presto 查询和计算 MySQL 中的数据时，Presto 只会启动单个线程读取表数据，当表的数据量巨大时，Presto 的工作效率变得低下甚至有可能导致查询失败。

PDBO: Presto for Database Optimizer, 针对 Presto 连接关系型数据库而开发的优化器，在 JD-Presto 版本中，Presto 读取关系型数据库的数据源涉及的相关功能统一被称为 PDBO，其特有的功能有：

- 多个节点读取单表，并能持续智能优化查询性能。
- 新增 SQL Server/Oracle 数据源。
- 支持更多字段类型的 Where 条件 Push Down 到 RDBMS。
- 支持 MySQL 分库分表。
- 可指定 Presto 计算节点读取表数据。
- 支持表数据缓存。

多个节点读取单表功能是指按照所设定的切分规则将表切分成若干段，每个 Presto 计算节点读取表的一部分数据，包含了动态步长计算和分批次读取功能。

动态步长优化：它是针对大表优化数据读取的一种处理模式。当多个 Worker 读取一张表的数据时，每个 Worker 读取一张表中的一部分数据时，可能由于表中数据稀疏不均，导致对表进行切分时各个 Worker 处理的数据不均匀而发生数据倾斜。PDBO 提供自动根据读取数据的日志智能计算每个 Worker 需要读取的数据量，从而保证各个 Worker 处理的数据量基本相等。

分批次读取：当接入 MySQL 分库分表功能后，分表的数量可能很大，一次性获取和传输所有分表的数据，可能对数据库服务器或者网络带宽造成很大的压力，从而影响线上环境的稳定，如何避免因数据读取给数据库和网络带来额外的压力是必须要解决的问题。PDBO 的分批次读取功能允许集群维护人员指定 Presto 每次读取指定个数的 Split，依次轮询读取所有的 Split，从而减轻数据库负载和网络压力。

新增 Oracle 和 SQL Server 数据源：Oracle 和 SQL Server 数据源均依赖于 presto-base-jdbc 模块，其实现原理和 presto-mysql 基本一致，其利用 JDBC 的方式从数据库中获取数据。

本章会为读者一步步揭开 PDBO 的神秘面纱。

14.1.1 JDBC Split 剖析

1. JDBC Split 基本原理

JDBC Split 是指在 Presto 读取关系型数据库数据源时，在数据读取阶段（source stage）形成的 Source Split，Presto 分配多少个 Worker 读取数据是基于 Source Split 的个数，也就决定了 Presto 读取数据的并发数。在 Presto master 版本中，每个表只会形成一个 Split，Split 相关信息封装在 com.facebook.presto.plugin.jdbc.JdbcSplit 中。如图 14-1 所示为 JdbcSplit 相关的类。

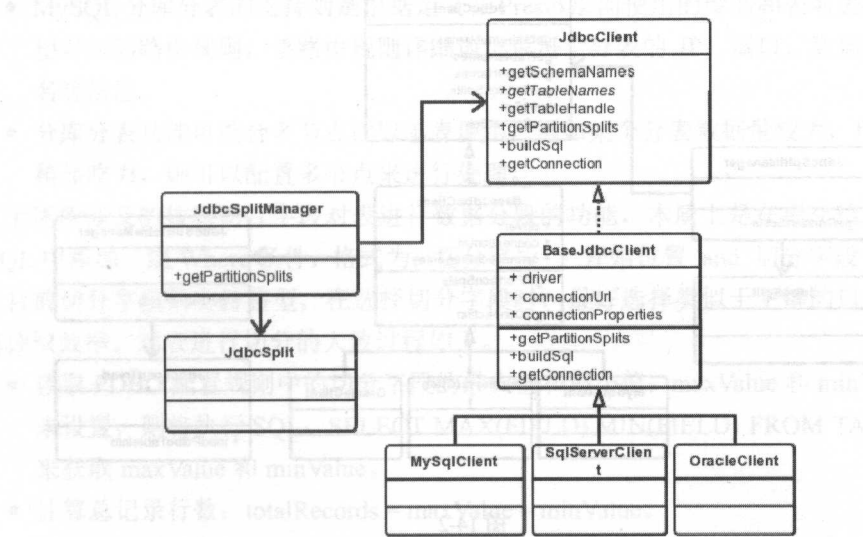


图 14-1

JDBC Split 获取通过 `JdbcSplitManagr.getPartitionSplits` 方法返回，具体构成是在 `com.facebook.presto.plugin.jdbc.BaseJdbcClient.getPartitionSplits(JdbcPartition jdbcPartition)` 方法中实现的，facebook master 版本代码如下：

```
@Override
public ConnectorSplitSource getPartitionSplits(JdbcPartition jdbcPartition)
{
    JdbcTableHandle jdbcTableHandle = jdbcPartition.getJdbcTableHandle();
    JdbcSplit jdbcSplit = new JdbcSplit(
        connectorId,
        jdbcTableHandle.getCatalogName(),
        jdbcTableHandle.getSchemaName(),
        jdbcTableHandle.getTableName(),
        connectionUrl,
        fromProperties(connectionProperties),
        jdbcPartition.getTupleDomain());
    return new FixedSplitSource(connectorId, ImmutableList.of(jdbcSplit));
}
```

JDBC Split 描述了查询表的详细信息，其中主要的信息包括 JDBC 连接信息以及查询字段和条件。JD-Presto 版本在 Catalog 配置文件中引入 `jdbc-sub-table-enable` 参数，参数值为 `true/false`，用于判断当前 Catalog 是否开启 PDBO 功能，若开启后，JDBC Split 将按照全新的逻辑进行构造。

2. PDBO Split 改造

新的 PDBO Split 对原 JDBC Split 进行了改造，并且引入新的 Split 生成逻辑，PDBO Split 的相关类如图 14-2 所示。

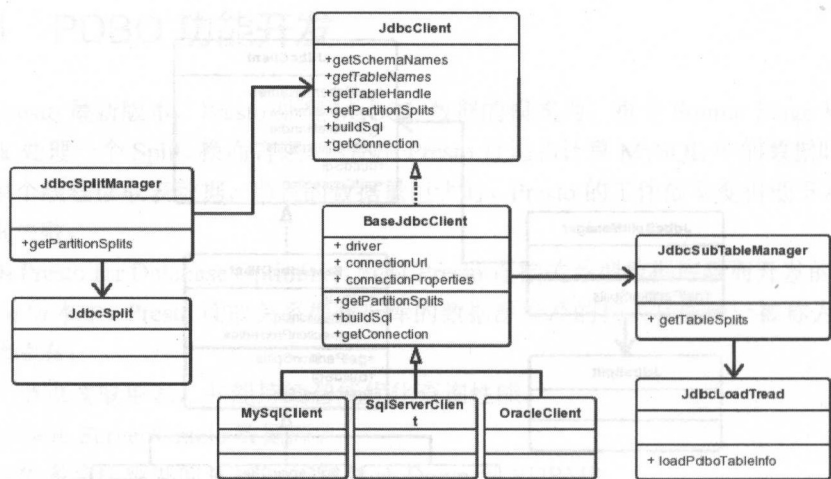


图 14-2

图 14-2 中的 JdbcSplitManager 根据 PDBO 定义的规则生成新的 Split, 每个 SQL 所涉及的关系型数据库的表都会经过新的逻辑重新定义读取数据规则, 其入口的代码为:

```
@Override
public ConnectorSplitSource getPartitionSplits(JdbcPartition jdbcPartition)
{
    if (jdbcSubTableEnable) {
        return subTableManager.getTableSplits(jdbcPartition);
    }
    .....
}
```

其中 subTableManager 为 JdbcSplitManager 实例对象, 是在 BaseJdbcClient 构造方法中进行初始化的, JdbcSplitManager 内部引用了 JdbcLoadThread 用于间隔的加载 PDBO 配置信息, 是在 JdbcSplitManager 的构造方法中进行创建的。

JdbcSplitManager.getTableSplits 方法的大致逻辑如下。

- 判断是否针对查询的表配置了 PDBO 规则, 若配置了 PDBO 规则, 则直接获取配置信息, 反之按照 Presto 原规则进行处理, 也就是以一个表一个 Split 的方式返回。
- 判断 PDBO 配置规则中是否启用了动态步长优化功能 (详见 14.1.4 节), 若启用了, 则获取优化计算完成的动态步长结果, 按照此结果来组织 Split。
- 读取数据节点数 (scanNodeNumber) 这项配置来决定 Split 的个数。如果配置了分库分表, 那么重构 Split 时将分表信息的连接信息 (包含了链接地址、用户名、密码等) 传入到 Split 实例中。
- 如果配置了多节点读取单表, 将根据配置中的切分字段相关信息来对表进行分段读取, 此时的分段信息严格按照数据偏移量来进行切分, 数据偏移量 = 总行数/总节点数。
- MySQL 分库分表的支持则是根据定义在 Presto 层面使用的库名和表名去查找是否有相对应的路由规则, 该路由规则详细地记录每个分表的 IP、端口、数据库、分表表名等信息。
- 分库分表功能可结合多节点读取单表使用, 例如某个分表数据量较大, 单 Split 处理稍显吃力, 则可以配置多节点来进行处理。

上述所涉及的按照切片字段对表进行数据分段的功能, 本质上是在提交给数据库执行的 SQL 中添加一组 Where 条件, 格式为: 切分字段 > 开始位置 and 切分字段 <= 结束位置。目前切分字段只支持整型, 在选择切分字段时, 最好选择类似于主键的自增字段, 可提高读取效率。对表进行切分的大致过程如下。

- 读取 PDBO 配置规则中的切分字段的最大值和最小值: maxValue 和 minValue。如果未设置, 则将执行 SQL: SELECT MAX(FIELD), MIN(FIELD) FROM TABLE; 语句来获取 maxValue 和 minValue。
- 计算总记录行数: totalRecords = max - min。
- 计算数据偏移量: offSet = totalRecords / scanNodeNumber。

- 根据切分字段组织 SQL 语句：`field > minValue and field <= (minValue + offset)`。

14.1.2 JdbcRecordCursor 剖析

1. 数据读取

Source Stage 会将 Split 通过 RESTful 接口远程调度到具体的 Presto Work 节点上，Work 节点根据 Split 描述的信息来获取数据，从整个 SQL 运行的角度来看，其大致过程如图 14-3 所示。

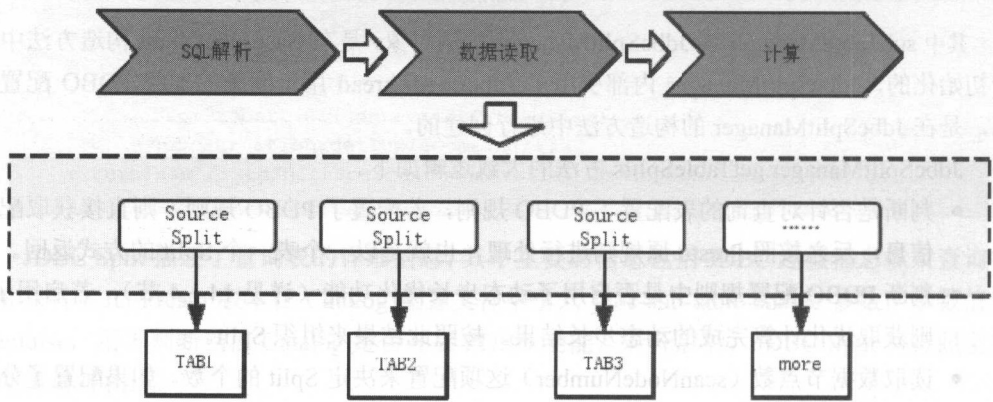


图 14-3

Presto 从发送连接请求到数据库读取数据都是由 JdbcRecordCursor 来完成的，在 JdbcRecordCursor 构造方法中根据 Split 信息来组织 SQL 语句，这个 SQL 语句是指需要从关系型数据库中获取数据的 SQL。获取数据的方式是通过 JDBC 来执行 SQL 语句并返回结果。Oracle 的 JDBC 驱动因 MAVEN 库中未提供，需要从 Oracle 官网下载；SQL Server 的 JDBC 驱动采用的是开源的 JTDS，GitHub 地址为：<https://github.com/milesibastos/jTDS>。图 14-4 展现了 JdbcRecordCursor 的相关类及其相互关系。

JdbcRecordCursor 本身的实现是相对简捷的，主要有以下几点。

- SQL 的组织调用了 BaseJdbcClient.buildSql。
- 每次读取数据前都会调用 advanceNextPosition 方法，判断是否还有下一条数据。
- Close 方法负责关闭 JDBC 连接。

在上面提到的根据切分字段形成 Where 条件（切分字段 > 开始位置 and 切分字段 <= 结束位置）会通过 Split 传到 JdbcRecordCursor 供其使用，在调用完 BaseJdbcClient.buildSql 后将会判断是否有切分条件，如果有则直接附加到 Where 条件中。

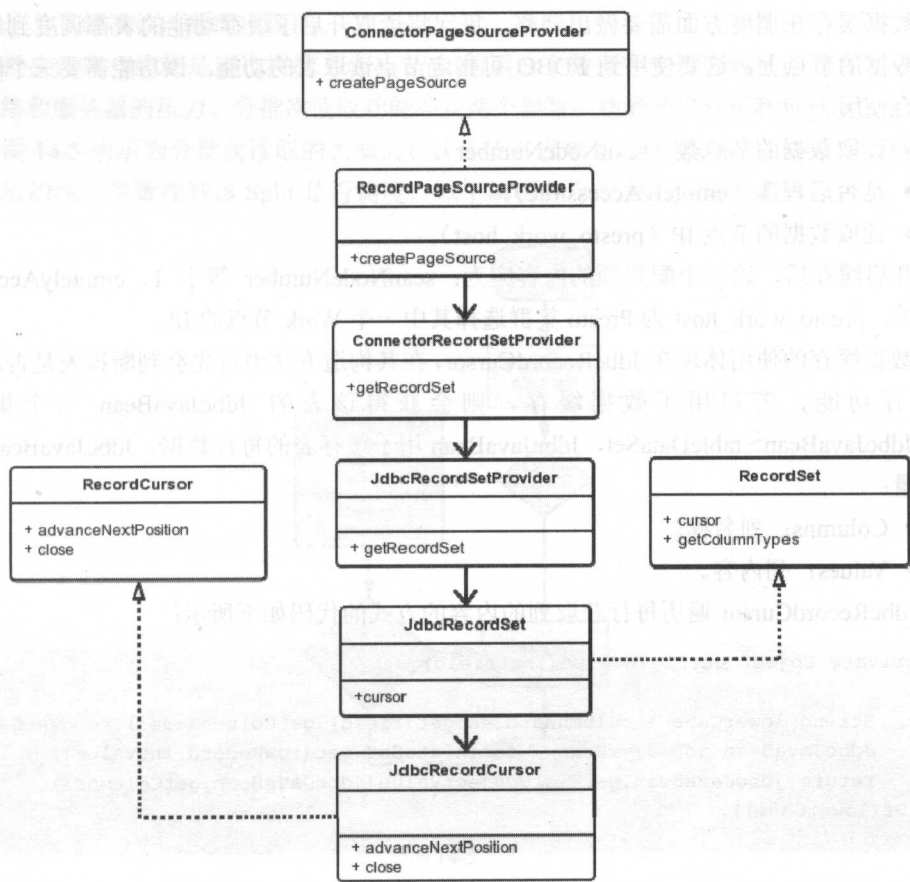


图 14-4

2. 数据缓存

数据缓存是指可将数据库表中的数据缓存到内存中，缓存策略使用的是 Guava Cache。数据缓存的本质是用空间换时间，当然也可以减少访问数据库的频率，降低数据库服务器的压力。数据缓存功能适用的场景大致有以下特点：

- 上层应用可以接受数据存在延迟的状态，不需要立刻查询最新数据。
- 查询条件相对固定。
- 缓存数据的节点有足够的内存来容纳需要缓存的数据。

缓存的相关配置均在 Catalog 配置文件中设置，涉及的参数主要有以下几个。

- jdbc-cache-enable：是否开启缓存功能，默认分为 false。
- jdbc-cache-table-config：配置需要缓存的表及其字段，内容为 json 格式：
[{"table_a":["code","create_time"]}, {"table_b":["id","code"]}]
- jdbc-cache-table-clause：配置缓存表的默认过滤条件，参数配置内包括表和对应的过滤条件，格式为 json: {"table_a":"create_time >= '2015-01-01 00:00:00',"table_b": "id >= 500000"}。

数据缓存在调度方面需要做出调整，每次将读取开启了缓存功能的表都调度到缓存了该表数据的节点上，这要使用到 PDBO 可指定节点读取表的功能，该功能需要三个配置项来配合使用。

- 读取数据的节点数 (scanNodeNumber)。
- 是否远程读 (emotelyAccessible)。
- 读取数据的节点 IP (presto_work_host)。

开启缓存后，这三个配置项的内容应为：scanNodeNumber 等于 1，emotelyAccessible 等于 N，presto_work_host 为 Presto 集群选择其中一个 Work 节点的 IP。

数据缓存的使用体现在 JdbcRecordCursor，在其构造方法中首先会判断该表是否启用数据缓存功能，若启用了数据缓存，则会获得该表的 JdbcJavaBean 一个集合：List<JdbcJavaBean> tableDataSet，JdbcJavaBean 用于缓存表的每行数据，JdbcJavaBean 有两个变量：

- Columns：列名称。
- Values：列内容。

JdbcRecordCursor 遍历每行获取列的内容的方式的代码如下所示：

```
private Object getFieldValue(int field)
{
    String lowerCase = columnHandles.get(field).getColumnName().toLowerCase();
    JdbcJavaBean jdbcJavaBean = tableDataSet.get(rowRecord.intValue() - 1);
    return jdbcJavaBean.getFieldObjectValue(jdbcJavaBean.getColumns().
indexOf(lowerCase));
}
```

14.1.3 分批次读取实现原理

数据读取需要网络传输，网络传输的快慢在很大程度上决定了整个 SQL 语句的执行速度，我们在配置 PDBO 多节点读取数据和 MySQL 分库分表时，可能会遇到以下情况：

- 配置的读取每张表的数据节点数 (scanNodeNumber) 比较大，比如 128。
- 配置的是 MySQL 分库分表，分表数量比较大，比如 1024 个。

如果没有设置分批次读取数据，那么 Presto 会一次性调度 PDBO 分配的所有 Split，那么会是什么的情况呢？

- 配置的读取某张表数据的节点数 (scanNodeNumber) 为 128，那么同时会向数据库发 128 个读该表的请求。
- 配置了 1024 个 MySQL 分表，并发地发送 1024 个读取请求。

当数据量比较大，或者还有别的数据库的操作时，以上两种状况可能会造成以下局面：

- 数据库负荷剧增，查询速度严重下降，极有可能导致数据库宕机。
- 网络带宽拥挤，容易造成超时，影响其他业务。

出现上述两种情况，对企业的生产环境来说都是极其危险的，此时急需某种策略可以根据当前数据库服务器压力，以及网络拓扑情况，对读取数据操作进行调整。分批次读取

功能就是为了解决该问题而出现的。

分批次读取功能就是通过控制 Source Split 的调度个数来控制读取数据库的并发量以减小对网络和服务器的压力。分批次读取功能涉及两个参数：功能开启标示和每批次的大小。

如图 14-5 所示为分批次读取的大概流程，其核心思想是在调度 Source Split 时，根据分批次读取的两个参数控制该 Split 是否调度，以下解析各个步骤的代码实现过程。

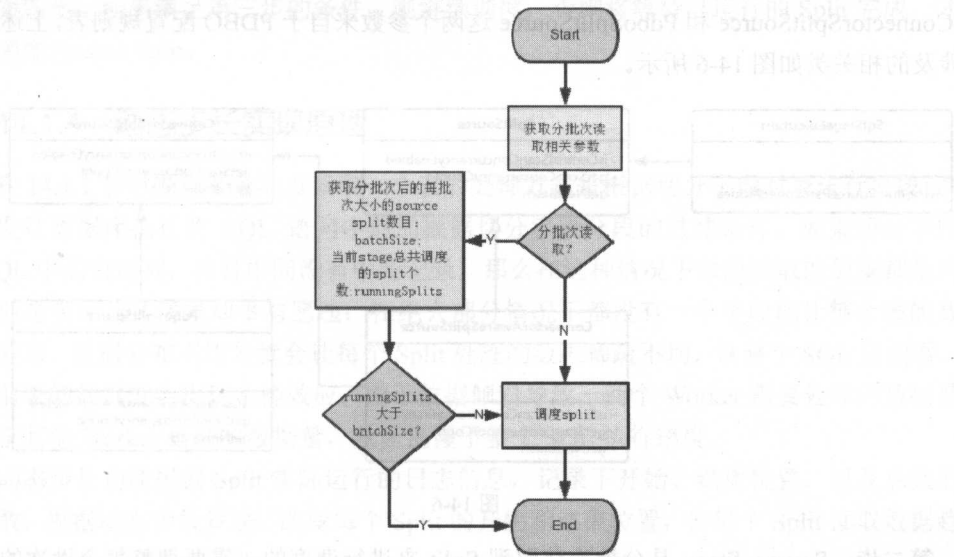


图 14-5

第一步：获取分批次读取相关参数。Source Split 的调度在 com.facebook.presto.execution.SqlStageExecution.scheduleSourcePartitionedNodes 方法中，通过 com.facebook.presto.split.SplitSource.getNextBatch(int maxSize)方法一次性获取到一批 Source Split 来进行调度，maxSize 的默认值为 1000。在 com.facebook.presto.split.SplitSource 接口中新增两个方法：

```
default boolean isControlScanConcurrencyEnabled()
{
    return false;
}

default int getScanConcurrencyCount()
{
    return 1;
}
```

SplitSource 实现类 com.facebook.presto.split.ConnectorAwareSplitSource 通过 com.facebook.presto.spi.ConnectorSplitSource 来重载这两个方法。

```
default boolean isControlScanConcurrencyEnabled()
{
    return false;
}
```

```
}

default int getScanConcurrencyCount()
{
    return 1;
}
```

新增的 `com.facebook.presto.plugin.jdbc.subtable.PdboSplitSource` 继承了 `com.facebook.presto.spi.ConnectorSplitSource` 和 `PdboSplitSource` 这两个参数来自于 PDBO 配置规则表, 上述过程所涉及的相关类如图 14-6 所示。

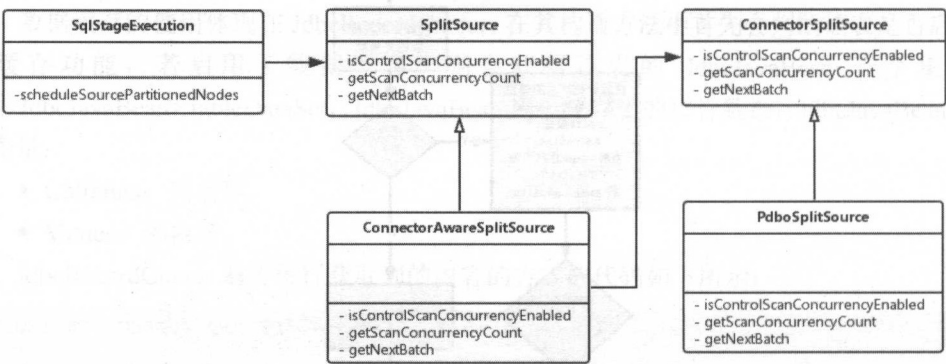


图 14-6

第二步: Source Stage 是分批次获取到 Split 来进行调度的, 需要调整每个批次的最大 Split 个数, 即为调整 `com.facebook.presto.split.SplitSource.getNextBatch(int maxSize)` 的 `maxSize` 参数:

```
boolean controlScanConcurrencyEnabled =
splitSource.isControlScanConcurrencyEnabled();
int scanConcurrencyCount = splitSource.getScanConcurrencyCount();
Set<Split> pendingSplits = ImmutableSet.copyOf(getFutureValue(
    splitSource.getNextBatch(controlScanConcurrencyEnabled ?
scanConcurrencyCount : splitBatchSize)));
```

第三步: 判断是否可以调度 JDBC Source Split。还是在 `SqlStageExecution.scheduleSourcePartitionedNodes` 方法中会调用 `com.facebook.presto.execution.NodeScheduler.computeAssignments` 方法, 该方法将 Split 分配给具体的 Worker 节点进行处理。首先在 `NodeScheduler.computeAssignments` 方法中计算出当前 Stage 已经运行的 Split 个数:

```
int allSplits = 0;
for (RemoteTask task : existingTasks) {
    .....
    allSplits += task.getPartitionedSplitCount();
}
```

然后会根据一系列的规则选择出可运行该 Split 的节点, 此时根据公式: 已分配 + 已运行 <= 分批次执行的最高上限, 判断是否调度该 Split, 代码如下:


```

if (chosenNode != null) {
    allSplits += 1;
    if (controlScanConcurrencyEnabled && scanConcurrencyCount < allSplits) {
        break;
    }
    .....
}

```

第四步：如果满足第三步的条件，则继续调度，否则将等待已运行的 Split 完成，才可继续调度 Source Split。

14.1.4 动态步长实现原理

在 14.1.1 节中提到多个节点读取一个表的处理方式是根据切分字段对表进行分段读取，最终发往数据库执行的 SQL 语句中追加根据切分字段分段的过滤条件。如果切分字段是 MySQL 中的自增列，并且中间没有中断记录，那么在这种情况下每段读取的数据都是均匀的。但是实际业务情景却事与愿违；在绝大部分情况下都没有一个字段能让整个表的数据均匀分布。数据分布不均匀就会让每个 Split 处理的数据稀疏不均，从整个 SQL 层面看，读取当前表的数据就会出现木桶效应，由于数据倾斜导致一两个 Worker 需要处理的数据量远远大于其他 Worker 处理的数据量，从而拖慢了整个 SQL 执行速度。

动态步长功能根据 Split 实际运行的日志信息，记录下开始、结束位置，以及总共的记录行数，根据动态步长算法，改变每个 Split 的开始和结束位置，使每个 Split 读取数据趋于平均，消除数据倾斜。

在 14.1.1 节介绍 PDBO 改造 Split 生成逻辑时提到，如果开启动态步长功能，则 JdbcRecordCursor 会将当期 Split 读取的开始位置、结束位置、总共读取行数记录到日志表中，根据动态步长算法分析日志，最终形成对表分段读取的优化结果，并根据优化结果来构造 JDBC Split，如图 14-7 所示。

图 14-7 描述的具体细节如下。

- 构造 JDBC Split 时获取 PDBO 配置规则，判断是否开启了动态步长的计算功能。
- 如果开启，则从 PDBO 日志表中查找状态为 finish 的记录，也就是经过动态步长算法计算后的 Split 分段详细信息；如果未开启，则默认按照固定步长的方式来重构 Split。
- 在每个 Split 读取结束后，JdbcRecordCursor.close()方法会调用日志写入功能，将该 Split 的行数、起始位置、结束位置等信息记录写入到日志表。
- Coordinator 会启用后台线程间隔性的检测是否有新的 PDBO 日志生成，若有新的日志生成，则根据动态步长算法对日志信息进行分析。

在 Coordinator 后台启动检测 PDBO 日志线程被称为动态步长算法管理线程，负责动态步长日志检测、算法调度、计算结果存储等。Coordinator 启动时，动态步长算法管理线程也随之启动，其基本功能如下。

- 管理线程会启动若干个用于计算动态步长的子线程，默认为 4 个，可配置。

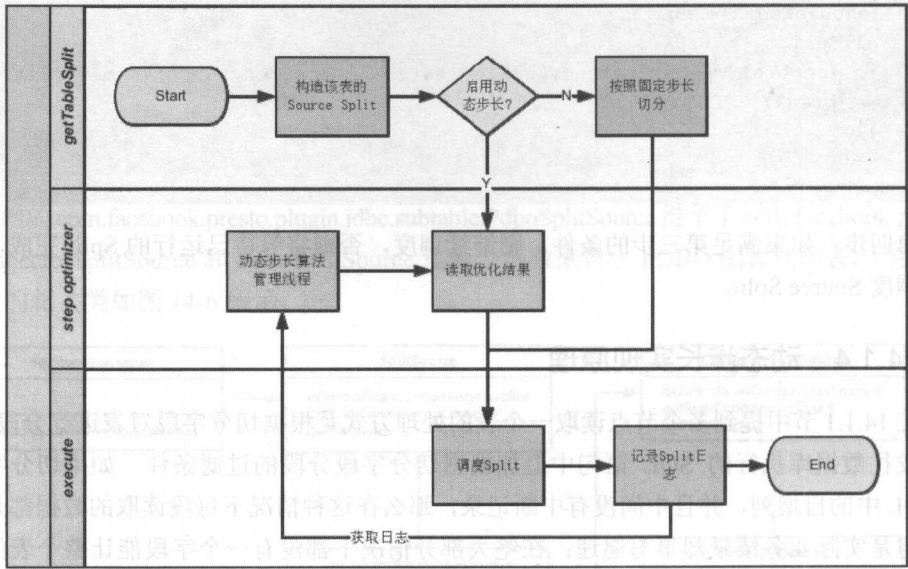


图 14-7

- 提取 PDBO 配置中启用了动态步长计算功能表的日志，并将该信息加入到全局排队列表中，可配置提取频率，默认为 3 分钟。
- 用于计算动态步长的子线程每次处理一个表信息的 PDBO 日志信息，处理时从全局排队列表中移除。
- 计算动态步长的子线程将计算完成的结果写入到 PDBO 日志表中，将原有日志状态标志为 history，新生成的计算结果状态标志为 finish。
- 对于历史日志信息，启用了清理历史数据的子线程，按照配置的间隔时间对历史数据进行清除，默认时长为 6 个小时。

在计算动态步长的子线程中定义了动态步长的计算算法。根据每条 PDBO 日志的总行数、起始位置、结束位置等信息计算出每个 Split 最优的步长。其算法的核心思想是：低于平均读取行数的进行合并，高于平均读取行数的进行拆分。算法的合并以及拆分过程详细介绍如下。

- 提取日志信息，并按照开始位置进行升序排序。
- 计算每个 Split 平均的读取行数： $averageStep = \text{sum}(\text{每个 Split 的总行数}) / \text{总节点数}$ 。
- 合并行数低于 $averageStep$ 的分片，并且记录所需要的节点数，合并时只能合并开始位置相邻的 Split。
- 计算出不需要拆分也不需要合并的分区，并且记录这些分区所需要的节点数。
- 将剩余的 Split 进行拆分处理，拆分的节点基数为：剩余的可用节点数目。每个剩余的 Split 需要拆分成多少个，其计算公式为： $(\text{Split 行数} / \text{剩余 Split 总行数}) \times \text{剩余可用节点}$ 。
- 拆分 Split 的步长： $SplitStep = (\text{Split 结束位置} - \text{Split 起始位置}) / \text{拆分个数}$ 。

- 最终的 Split 的个数为：合并后的个数 + 未变化的个数 + 拆分后的个数。

14.1.5 条件下发

1. 为什么需要条件下发

Presto 的 JDBC 连接器通过 JDBC 连接关系型数据库（MySQL、PostgreSQL 等），读取关系型数据库的数据到 Presto 集群中进行后续的处理。很明显，读取的数据量越多，占用的内存就会越多，处理速度也会越慢，因此需要尽可能减少 Presto 从关系型数据库中读取的数据量。

2. 现状

对于涉及多个表的比较复杂的 SQL 语句，Presto 对其进行解析后会进行拆分，组成多个针对不同表的简单的 Select 语句，如：

```
SELECT x FROM A
```

如果原 SQL 语句中有 Where 条件，则同样会将该条件下发，如：

```
SELECT x FROM A WHERE x < 10
```

对 Presto 的 Jdbc Connector 来说，目前只支持 Bigint 类型、Double 类型以及 Boolean 类型的 Where 条件下发，并不支持 Varchar 类型、Date 类型和 TimesTamp 类型的 Where 条件下发。

3. 完善条件下发

通过修改 Presto JDBC 连接器可以添加对 Varchar 类型、Date 类型和 TimesTamp 类型的 Where 条件下发。

(1) Varchar 类型

Presto 的 Varchar 类型在内存中是用 Slice 来进行处理的，Slice 是 Presto 开发成员维护的 RESTful 框架 Airlift 的子项目，用于实现类 C 语言效率的内存拷贝。

在拼接传递给 JDBC Connector 的 SQL 语句时，使用以下方法获取 Where 条件中 Varchar 类型的值：

```
private static String encode(Object value, Type columnType)
{
    ...
    if (value instanceof Slice) {
        return "'" + ((Slice) value).toStringUtf8() + "'";
    }
    ...
}
```

传递给 JDBC Connector 的 SQL 语句将是如下形式：

```
SELECT x FROM A WHERE x = 'Hello'
```

(2) Date 类型

Presto 的 Date 类型在内存中是以 Java 的 Long 类型进行处理的,存储的是从 1970-01-01 到指定时间的天数,在 1970-01-01 之后为正数,在 1970-01-01 之前为负数。

在拼接传递给 JDBC Connector 的 SQL 语句时,获取 Where 条件中 DATE 类型的值,并转成“yyyy-MM-dd”格式的字符串:

```
if (columnType.equals(DateType.DATE)) {
    return "" + new SimpleDateFormat("yyyy-MM-dd").format(new Date(86400000 *
Long.parseLong(value.toString(), 10))) + "";
}
```

传递给 JDBC Connector 的 SQL 语句将是如下形式:

```
SELECT x FROM A WHERE x = '2015-06-18'
```

(3) TimesTamp 类型

Presto 的 TimesTamp 类型在内存中同样是以 Java 的 Long 类型进行处理的,存储的是从 1970-01-01 00:00:00 到指定时间的毫秒数,在 1970-01-01 00:00:00 之后为正数,在 1970-01-01 00:00:00 之前为负数。

在拼接传递给 JDBC Connector 的 SQL 时,获取 Where 条件中 TimesTamp 类型的值,并转成“yyyy-MM-dd HH:mm:ss.SSS”格式的字符串:

```
if (columnType.equals(TimestampType.TIMESTAMP)) {
    return "" + new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new
Date(Long.parseLong(value.toString(), 10))) + "";
}
```

传递给 JDBC Connector 的 SQL 将是如下形式:

```
SELECT x FROM A WHERE x = '2015-06-18 00:00:00.000'
```

(4) 性能提升

针对一个连接关系数据库的复杂 SQL,其中 Where 条件中有 DATE 类型,对比测试结果如图 14-8 所示。

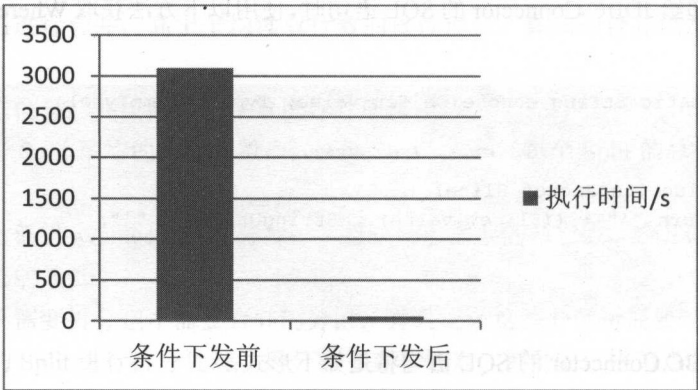


图 14-8

可见，完善条件下发后 SQL 执行时间缩短了数十倍，大大提高了执行效率。

14.1.6 PDBO 配置定义

从上述章节可知，PDBO 功能需要依赖相关的配置项，PDBO 的配置分为两部分：

- Presto Catalog 中新增的配置参数。
- 详细的 PDBO 规则配置都保存在 MySQL 中。

1. PDBO 参数配置

要开启 PDBO 相关功能，则需要在对应的 Catalog 配置文件中配置如表 14-1 所示的参数。

表 14-1

参数名称	说 明	默认值
jdbc-sub-table-connection-url	分表信息表数据库实例地址	
jdbc-sub-table-connection-user	连接分表信息表用户	
jdbc-sub-table-connection-password	连接分表信息表密码	
jdbc-reload-subtable-interval	加载分表信息的时间间隔	5min
jdbc-sub-table-enable	是否启用分表功能标示	false

例如，一个 MySQL Connector 的 mysql_test.properties 可进行以下配置：

```
connector.name=mysql
connection-url=jdbc:mysql://127.0.0.1:3306
connection-user=root
connection-password=root
#pdbo configuration
jdbc-reload-subtable-interval=5m
jdbc-sub-table-connection-url=jdbc:mysql://127.0.0.1:3306
jdbc-sub-table-connection-user=user
jdbc-sub-table-connection-password=password
jdbc-sub-table-enable=true
```

2. PDBO 表定义（见表 14-2）

表 14-2

Pdbo 表信息: Pdbo_table			
字 段	字段名称	类 型	说 明
Tableid	主键	bigint	自增
Dbtype	数据库类型	Varchar	MySQL SQL Server Oracle

续表

字 段	字段名称	类 型	说 明
Connectorid	Presto catalog 连接标示符	varchar(50)	必须和 Presto 中定义 properties 配置文件名称前缀一致。比如 Presto 中定义 MySQL 数据源为: coo.properties 配置文件, 那么此字段配置内容为: coo
presto_schema	Presto schema 名称	varchar(50)	由存有表结构的数据库实例中的所有的 database 的名称来决定
presto_table	Presto 中使用的查询表名	varchar(50)	由存有表结构的数据库实例中的所有的 table 的名称来决定
calc_step_enable	开启动态步长计算标示	char(1)	默认为 N
control_scan_concurrency_enabled	并发控制标示	char(1)	抓取表数据时, 是否开启并发控制, 默认为 N
scan_concurrency_count	并发上限值	int	当并发控制功能开启时, 设置并发的数量

3. PDBO 路由表定义 (见表 14-3)

表 14-3

Pdbo 路由信息: Pdbo_route			
字 段	字段名称	类 型	说 明
Routeid	主键	bigint	自增
tableid	Pdbo_table 主键	bigint	
dbhost	数据库 IP	varchar(25)	存储实际数据的数据库实例 IP
dbport	数据库端口	varchar(10)	存储实际数据的数据库实例端口
connection_properties	连接参数	varchar(200)	JDBC 连接参数
source_schema	数据库名称	varchar(50)	存储实际数据的数据库实例上的 database 名称
source_table	表名称	varchar(50)	存储实际数据的数据库实例上的 table 名称
splitfield	切分字段	varchar(50)	切分字段是为了使 Presto 节点分段读取表, 配置内容为表字段名, 暂时支持的字段类型为整型和日期型, 最终会在查询数据的 SQL 的 Where 条件中添加类似于如下的过滤条件: splitfield > xx and splitfield <= xxxx。如果 scannodenumbr 所配置的值大于 1, 切分字段必须配置
fieldmaxvalue	切分字段最大值	bigint	如果设置了切分字段, 必须设置切分字段最大值, 且必须大于 0
fieldminvalue	切分字段最小值	bigint	默认为 0
remotelyaccessible	是否远程读	char(1)	是否远程读
presto_work_host	Presto 计算节点	varchar(25)	指定 Presto 节点读取该表数据, 如果设置了切分字段, 那么所重构的 Split 的 HostAddress 均为该 Presto 节点。
scannodenumbr	读取数据的节点数量	int	读取该表数据的节点数, 也可以理解为读取该表数据的线程数。可针对每个表进行设置。
uid	数据源信息关联字段	varchar(50)	

4. 数据库信息表（见表 14-4）

表 14-4

数据源信息表：db_info			
字 段	字段名称	类 型	说 明
Id	主键	int	
dbhost	MySQL 实例 IP		
dbport	MySQL 实例端口		
uid	与 table_route 关联字段		
username	用户名		
password	密码		

5. 建表语句

```
CREATE TABLE `pdbo_table` (  
  `tableid` bigint(20) NOT NULL AUTO_INCREMENT,  
  `dbtype` enum('MYSQL','SQLSERVER','ORACLE') NOT NULL,  
  `connectorid` varchar(50) DEFAULT NULL,  
  `presto_schema` varchar(50) DEFAULT NULL,  
  `presto_table` varchar(50) DEFAULT NULL,  
  `calc_step_enable` char(1) NOT NULL DEFAULT 'N',  
  `control_scan_concurrency_enabled` char(1) NOT NULL DEFAULT 'N',  
  `scan_concurrency_count` int(11) DEFAULT NULL,  
  PRIMARY KEY (`tableid`)  
);  
  
CREATE TABLE `pdbo_route` (  
  `routeid` bigint(20) NOT NULL AUTO_INCREMENT,  
  `tableid` bigint(20) NOT NULL,  
  `dbhost` varchar(25) DEFAULT NULL,  
  `dbport` varchar(10) DEFAULT NULL,  
  `connection_properties` varchar(200) DEFAULT NULL,  
  `source_schema` varchar(50) DEFAULT NULL,  
  `source_table` varchar(50) DEFAULT NULL,  
  `splitfield` varchar(50) DEFAULT NULL,  
  `remotelyaccessible` char(1) NOT NULL DEFAULT 'Y',  
  `presto_work_host` varchar(25) DEFAULT NULL,  
  `scannodenummer` int(11) DEFAULT NULL,  
  `uid` varchar(50) DEFAULT NULL,  
  `fieldmaxvalue` bigint(20) DEFAULT NULL,  
  `fieldminvalue` bigint(20) DEFAULT NULL,  
  PRIMARY KEY (`routeid`)  
);  
  
CREATE TABLE `db_info` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `dbhost` varchar(50) DEFAULT NULL,  
  `dbport` varchar(10) DEFAULT NULL,  
  `uid` varchar(50) DEFAULT NULL,  
  `username` varchar(30) DEFAULT NULL,  
  `password` varchar(50) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
);
```

14.2 DDL 及 DML 支持

截止到 0.107 版本, Presto 的 Hive 连接器支持的 DDL 和 DML 操作较少, 只有 `create table` 和 `create table as select` (CTAS)。而在实际应用过程中很多时候都需要 `insert` 操作, 虽然可以通过 Presto 的 CTAS 在 Hive 中建临时表, 然后在 Hive 中执行 `Insert`, 操作起来却比较复杂, 因此为 Hive 连接器实现常用的 DDL 和 DML 操作很有必要。

14.2.1 Hive 连接器 Insert 功能

Hive 连接器的 `Insert` 功能实现参考了 Presto 原有的 `Create Table as Select` (CTAS) 功能, 主要实现流程如下。

1. 词法语法解析

- 修改语法定义文件 `SqlBase.g4`

```
INSERT INTO qualifiedName query #insertInto
```

修改为:

```
INSERT (INTO | OVERWRITE) (TABLE)? qualifiedName
      (PARTITION '(' partitionDef (',' partitionDef)* ')')? query #insert
```

其中 `partitionDef` 的定义为:

```
partitionDef
    : identifier (EQ valueExpression)? # partitionElement
    ;
```

- 添加 `PartitionElement` 类

该类表示 SQL 语句中指定的分区和对应值, 主要包含以下元素:

```
private final String name; // 分区名
private final Optional<Expression> value; // 分区值, 动态分区时为空
```

- 修改 `Insert` 类

添加如下三项:

```
private final boolean overwrite; // 是否有 overwrite 关键字
private final boolean partition; // 是否有 partition 关键字
private final List<PartitionElement> partitionElements; // 语句中的分区及对应值
```

- 修改 `AstBuilder`

```
public Node visitInsertInto(@NotNull SqlBaseParser.InsertIntoContext context)
{
    return new Insert(getQualified_name(context.qualifiedName()), (Query)
visit(context.query()));
}
```

修改为:


```
public Node visitInsert(@NotNull SqlBaseParser.InsertContext context)
{
    return new Insert(getQualifiedName(context.qualifiedName()),
        (Query) visit(context.query()),
        context.OVERWRITE() != null,
        context.PARTITION() != null,
        visit(context.partitionDef(), PartitionElement.class));
}
```

并添加方法：

```
public Node visitPartitionElement(@NotNull SqlBaseParser.
PartitionElementContext context)
{
    Optional<Expression> value = Optional.empty();
    if (context.EQ() != null) {
        value = Optional.of((Expression) visit(context.valueExpression()));
    }
    return new PartitionElement(context.identifier().getText(), value);
}
```

2. 语义分析

- 添加类 `InsertOption`，用于记录对 `Insert` 分析的结果。

该类主要元素有：

```
private final boolean overwrite; // 是否是 overwrite
private final boolean isPartition; // 是否是插入分区表
private final boolean dynamicPartition; // 是否是动态分区
private final LinkedHashMap<String, String> partitions; // 分区列及对应值
```

- 修改 `StatementAnalyzer`

在 `visitInsert` 方法中添加：

```
analyzeInsertOption(insert, columns, targetTable);
```

主要用于分析语句是否是动态分区，分区名与分区值的对应关系，分区是否有效，以及一些异常处理，最后构造一个 `InsertOption` 记录下来。

3. 执行计划生成

- 用 `TableWriterNode` 的 `InsertReference` 类添加类成员 `InsertOption`。
- 用 `Metadata` 的 `beginInsert` 方法添加一个参数 `InsertOption`。

4. Hive Connector

- 添加 `PartitionOption` 类

`PartitionOption` 类用于记录与分区相关的一些参数，其类成员有：

```
private final boolean dynamicPartition; // 是否是动态分区
private final LinkedHashMap<String, String> partitionElements; // 分区列及对应值
private final String partitionSuffix; // 分区名的 hdfs 后缀
```

- 添加 HiveInsertTableHandle 类

HiveInsertTableHandle 类与 HiveOutputTableHandle 类似, 不同的几个类成员为:

```
private final List<Boolean> partitionBitmap; // 表示该表的某一列是否是分区列
private final String filePrefix;           // 一个 UUID, 用于临时目录命名
private final PartitionOption partitionOption; // 分区相关参数
private final boolean overwrite; // 是否是 overwrite
```

- 修改 HiveMetadata

参考 beginCreateTable 和 commitCreateTable 方法, 实现 beginInsert 和 commitInsert 方法。

beginInsert: 构造 PartitionOption 和 HiveInsertTableHandle。

commitInsert: 移动写入的数据到目的表, 依据 HiveInsertTableHandle 为目的表添加分区, 如果是 overwrite, 则需先删除原有对应分区。

- 修改 HiveRecordSink

记录写入数据的分区, 为分区创建目录并将数据写入对应分区。

5. 实现功能

按照以上流程实现的 Hive 连接器的 insert 包含以下功能。

- insert into 或 insert overwrite 未分区表

```
insert (into | overwrite) table_name select ...
```

- insert into 或 insert overwrite 静态分区

```
insert (into | overwrite) table_name partition(dt='1970-01-01') select ...
```

- insert into 或 insert overwrite 动态分区

```
insert (into | overwrite) table_name partition(dt) select ..., dt from ...
```

14.2.2 Hive 连接器 CTAS 动态分区表功能

该功能语法如下:

```
create table table_name partition (dt) as select ..., dt from ...
```

该功能不是标准的 SQL 语法, 开发该功能是为了方便从其他的连接器向 Hive 连接器中导数。当使用该语法后, 可以在 Hive 连接器中根据查询的数据直接建立动态分区表, 无须在 Hive 中建立分区表再在 Presto 中执行 Insert, 方便操作。

该功能实现流程如下。

1. 词法语法解析

- 修改语法定义文件 SqlBase.g4

```
CREATE TABLE qualifiedName AS query #createTableAsSelect
```

修改为:

```
CREATE TABLE qualifiedName
    (PARTITION '(' IDENTIFIER (',' IDENTIFIER)* ')')? AS query
#createTableAsSelect
```

- 添加 CreateTableOption 类

该类包含了以下类成员：

```
private final boolean isPartition; // 是否是 CTAS 动态分区表
private final List<String> partitions; // 动态分区的列
```

- 修改 CreateTableAsSelect 类

添加类成员：

```
private final boolean isPartition; // 是否是 CTAS 动态分区表
private final List<String> partitions; // 动态分区的列
```

- 修改 AstBuilder

visitCreateTableAsSelect 方法修改为：

```
public Node visitCreateTableAsSelect(@NotNull
SqlBaseParser.CreateTableAsSelectContext context)
{
    List<String> columns = context.IDENTIFIER().stream()
        .map(ParseTree::getText)
        .collect(Collectors.toList());
    return new CreateTableAsSelect(getQualifiedName(context.qualifiedName()),
(Query) visit(context.query()), context.PARTITION() != null, columns);
}
```

2. 语义分析

- 修改 StatementAnalyzer

visitCreateTableAsSelect 方法添加以下代码：

```
// 处理分区相关的异常
if (node.isPartition()) {
    Iterable<Optional<String>> queryColumns =
transform(descriptor.getVisibleFields(), Field::getName);
    for (String partitionColumn : node.getPartitionList()) {
        if (!contains(queryColumns, Optional.of(partitionColumn))) {
            throw new SemanticException(MISSING_PARTITION, node, "Target table
'%s' is partitioned but query does not specifys the partition", targetTable);
        }
    }
    analysis.setCreateTableOption(new CreateTableOption(node.isPartition(),
node.getPartitionList()));
}
```

3. 执行计划生成

- 修改 LogicalPlanner

getOutputTableColumns 方法，修改为：

```
private static List<ColumnMetadata> getOutputTableColumns(RelationPlan plan,
CreateTableOption createTableOption)
{
    ImmutableList.Builder<ColumnMetadata> columns = ImmutableList.builder();
```

```

boolean isPartition = createTableOption.isPartition();
List<String> partitionColumns = createTableOption.getPartitions();
for (Field field : plan.getDescriptor().getVisibleFields()) {
    // 确定分区列
    boolean isColumnPartitioned = false;
    if (isPartition && partitionColumns.contains(field.getName().get())) {
        isColumnPartitioned = true;
    }
    columns.add(new ColumnMetadata(field.getName().get(), field.getType(),
isColumnPartitioned));
}
return columns.build();
}

```

4. Hive Connector

- 修改 HiveOutputTableHandle

添加类成员：

```
private final List<Boolean> columnPartitioned; // 表示该表的某一列是否是分区列
```

- 修改 HiveMetadata

beginCreateTable 方法在构造 HiveOutputTableHandle 时，同时记录下哪些列是分区列。
commitCreateTable 方法根据写入的分区文件夹添加动态分区。

- 修改 HiveRecordSink

与以上 Insert 动态分区的操作类似。

14.3 动态增加、修改、删除 Catalog

14.3.1 目的

在实际使用 Presto 的过程中，经常会有以下的一些需求。

- 添加一个新的 Catalog。
- 对不再使用的 Catalog 希望把它删除。
- 修改某个 Catalog 的参数。

但在 Presto 中如果进行上述的修改，需要重启 Presto 服务才可生效，这给集群维护带来额外的工作量之外，还给上层应用带来很不好的使用体验。因此本章节将详细介绍如何实现动态增加、修改、删除 Catalog 的功能。

14.3.2 现状

Presto 目前加载 Catalog 的策略如图 14-9 所示。

在 Presto 节点启动完成后会周期性地向 Coordinator 汇报其所拥有的 Catalog 信息，在 Coordinator 分配任务时使用。可以看到，目前 Presto 是在节点启动时一次性加载 Catalog，之后 Catalog 的信息会常驻内存中直至节点停止运行。而在节点运行期间 Catalog 信息是无

法被增加、删除或修改的，如果需要增加、删除、修改 Catalog 就必须重启节点。然而在集群运行过程中重启节点甚至整个集群是不允许重启节点的，因此需要一种更加灵活的方式来动态增加、删除、修改 Catalog，而无须重启节点。

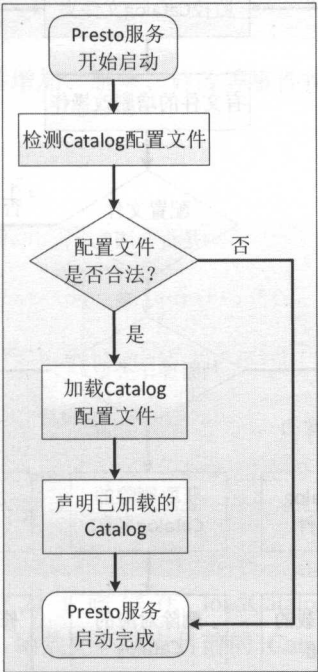


图 14-9

14.3.3 实现

实现动态增加、删除、修改 Catalog 的主要流程为：

- (1) Presto 服务启动后开启一个后台监控线程，监控 Catalog 配置文件目录。
- (2) 当 Catalog 配置文件目录中有文件的增加、修改、删除操作时，检测有变动的配置文件，并在 Presto 服务中进行相应的增加、删除、修改 Catalog 操作。

当监控线程检测到 Catalog 配置文件目录中有文件的增删改操作时，根据操作类型的不同在 Presto 服务中进行相应的操作。

- 当增加配置文件时，判断新增的配置文件是否合法，如果合法，则加载到 Presto 服务中，并且更新节点声明，将新增的 Catalog 加入自身所拥有的合法的 Catalog 列表中。
- 当删除配置文件时，获取删除的 Catalog 名，将 Presto 服务中对应的 Catalog 删除，并且更新节点声明，将删除的 Catalog 从自身所拥有的合法的 Catalog 列表中删除。
- 当修改配置文件时，加载修改的配置文件，修改 Presto 服务中对应的 Catalog 配置信息，并且更新节点声明，更新自身所拥有的合法的 Catalog 列表。

具体实现流程如图 14-10 所示。

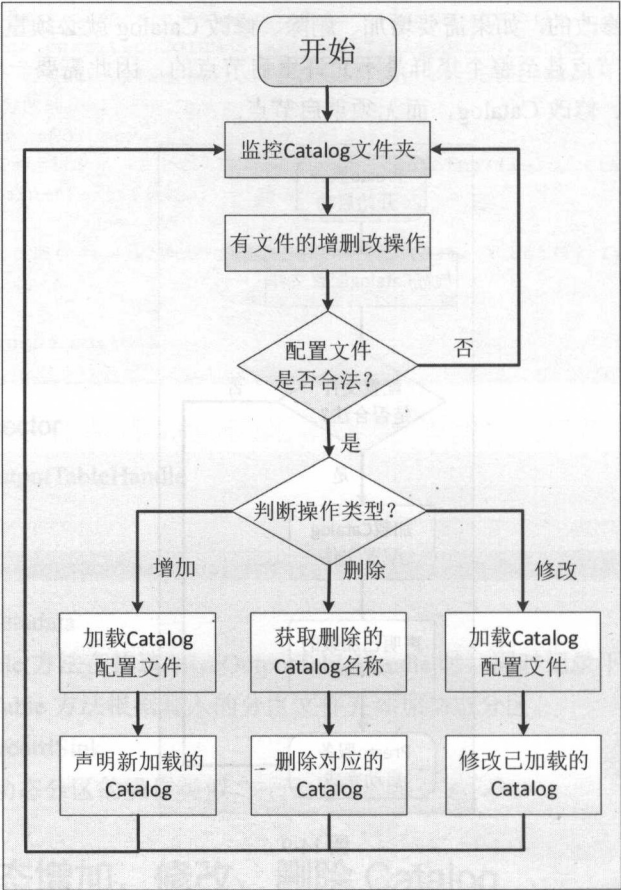


图 14-10

1. Catalog 监控线程

Catalog 监控线程使用 `java.nio.file.WatchService` 实现，监控线程的主要代码如下所示：

```
WatchService watchService = FileSystems.getDefault().newWatchService();
Paths.get(catalogConfigurationDir.getAbsolutePath()).register(
    watchService, StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_DELETE,
    StandardWatchEventKinds.ENTRY_MODIFY);
while (true) {
    WatchKey key = watchService.take();
    for (WatchEvent<?> event : key.pollEvents()) {
        if (event.kind() == StandardWatchEventKinds.ENTRY_CREATE) {
            Path newCatalog = (Path) event.context();
            addCatalog(newCatalog);
        }
        else if (event.kind() == StandardWatchEventKinds.ENTRY_DELETE) {
            Path deletedCatalog = (Path) event.context();
            deleteCatalog(deletedCatalog);
        }
    }
}
```

```

else if (event.kind() == StandardWatchEventKinds.ENTRY_MODIFY) {
    Path modifiedCatalog = (Path) event.context();
    modifyCatalog(modifiedCatalog);
}
}
....

```

监控线程根据检测到的文件增加、删除、修改等事件执行不同的操作。

2. 增加 Catalog

增加 Catalog 的主要代码如下：

```

private void addCatalog(Path catalogPath)
{
    File file = new File(catalogConfigurationDir, catalogPath.getFileName().
toString());
    if (file.isFile() && file.getName().endsWith(".properties")) {
        try {
            TimeUnit.SECONDS.sleep(5);
            loadCatalog(file);
            updateDatasourcesAnnouncement(Files.getNameWithoutExtension
(catalogPath.getFileName().toString()), DatasourceAction.ADD);
        }
    }
    ...
}

```

其中事先 sleep 5 秒是为了确保在 loadCatalog 时文件已经完全写入，updateDatasourcesAnnouncement 函数更新节点目前的 Catalog 信息声明，具体实现参考类 TestingPrestoServer 的同名方法。

3. 删除 Catalog

增加 Catalog 的主要代码如下：

```

private void deleteCatalog(Path catalogPath)
{
    if (catalogPath.getFileName().toString().endsWith(".properties")) {
        String catalogName =
Files.getNameWithoutExtension(catalogPath.getFileName().toString());
        connectorManager.removeConnector(catalogName);
        updateDatasourcesAnnouncement(catalogName, DatasourceAction.DELETE);
    }
    ...
}

```

其中 connectorManager.removeConnector(catalogName)从 Catalog 列表和相关的系统表以及数据结构中移除，参考 connectorManager.addConnector 中的系统表和数据结构。

updateDatasourcesAnnouncement 同样参考类 TestingPrestoServer 的同名方法。

4. 修改 Catalog

修改 Catalog 的代码如下：

```

private void modifyCatalog(Path catalogPath)
{
    deleteCatalog(catalogPath);
}

```

```
addCatalog(catalogPath);  
}
```

可见修改 Catalog 的操作在实际执行时是先删除 Catalog, 然后再添加修改后的 Catalog。

14.3.4 效果

实现了 Catalog 的动态增加、删除、修改之后, 使得 Presto 集群的管理更加灵活与方便, 无须频繁重启集群, 可以大大减轻运维人员的负担。

14.4 小结

根据京东自身的业务要求, 京东 Presto 版本着重从以下几点进行了改造。

- PDBO 模块, 针对 Presto 连接关系型数据库进行优化, 使 Presto 连接关系型数据源时可具有更高效的查询效率, 以及更广泛的使用场景。
- 京东使用 Hive 由来已久, 分区表以及 Insert 等相关操作使用非常频繁, 为了使业务迁移更加方便以及降低 Presto 学习成本, 实现了 Insert 相关功能。
- Presto 增加或者删除 Catalog 需要重启集群, 为了更加方便快捷地维护集群, 动态加载 Catalog 是非常必要的。

本章对以上列举的几点从工作原理到详细的源码实现都进行了深入浅出的讲解, 全面介绍了京东 Presto 版本的功能改造。

第 15 章

Presto 性能调优

我们在使用 Presto 过程中，会涉及多种应用场景，针对不同的应用场景，可以采用不同的措施提升 Presto 的性能。本章的主要内容就是针对不同的应用场景，详细讲述 Presto 性能优化的手段。

15.1 合理设计分区

本节会以读取 Hive 数据仓库中的数据为例，详细讲解在使用 Presto 查询 Hive 中的数据时，如何合理地使用分区。

Hive 数据仓库中的数据可以通过某些表示时间、区域等信息的字段进行分区，使用 Presto 查询 Hive 数据仓库中的数据时，通过在查询语句中根据分区字段限定查询的范围，可以有效地避免读取不必要的数据。这样可以明显地提升查询的性能。

15.2 Group By 字句优化

还有一个可以将 Presto 的查询性能稍微提升一点的方式就是：合理安排 Group By 字句中字段的顺序。若 Group By 字句中有两个字段，其中 Column1 中的 distinct 值的数量要远远多于字段 Column2 中 distinct 值的数量，则需要将 Column1 放在 Column2 的左边。也就是将 Group By 字句中的字段按照每个字段 distinct 数目的多少进行降序排列。例如：

```
[GOOD]: SELECT count(*),userid,country from employees GROUP BY userid, country  
[BAD]:  SELECT count(*),userid,country from employees GROUP BY country, userid
```

15.3 使用模糊聚合函数

Presto 提供了一系列的模糊聚合函数，使用这些函数虽然会有一点误差，但是可以获得

巨大的性能提升。例如使用模糊聚合函数 `approx_distinct()` 函数，你的查询性能相对于 `count(distinct x)` 来说会获得巨大的提升，而且与 `count(distinct x)` 查询的结果的误差率为 2.3%。下面的语句是查询京东商城某一天登录网站的用户数。

```
SELECT
  approx_distinct(user_id)
FROM
  jd_login
WHERE
  login_date='2015-09-16'
```

15.4 合并多条 Like 子句为一条 regexp_like 子句

Presto 的查询优化器无法对使用很多 Like 字句的查询进行优化。因此，若你的查询语句中还有很多 Like 子句，那么你的查询执行的性能会非常低下。

为了提升性能，可以将以 OR 语句连接的多个 Like 子句写成一个 `regexp_like` 子句。例如：

```
SELECT
  ...
FROM
  jd_login
WHERE
  city LIKE '%BEIJING%' OR
  city LIKE '%SHANGHAI%' OR
  city LIKE '%HANGZHOU%' OR
  city LIKE '%GUANGZHOU%'
```

可以对上面的语句进行优化：将四个 Like 子句以一个单独的 `regexp_like` 子句进行表示：

```
SELECT
  ...
FROM
  jd_login
WHERE
  regexp_like(city, 'BEIJING|SHANGHAI|HANGZHOU|GUANGZHOU')
```

15.5 大表放在 Join 子句左边

在默认情况下，Presto 使用 Distributed Hash Join 算法，在这种算法中，Join 左右两边的表都会根据 join 键的值进行分区。左表的每个分区都会被通过网络传入到右表所在分区的 Worker 节点上。也就是说，在进行 Join 关联的时候，会先把右表的所有分区全部分布到各个计算节点上保存，然后等待将左表中的各个分区依次通过网络传输 Stream 到相应的计算节点上进行计算。由于右表的所有分区需要全部分布到各个节点上进行存储，所以有一个限制：就是集群中所有内存的大小一定要大于右表的大小。

如果你在执行 join 查询的时候看到错误: Task exceeded max memory size, 那么这经常意味着 join 连接的右表大于集群所有内存的大小。Presto 不会自动进行 join 两边表顺序优化, 因此执行 Join 查询的时候, 请确保大表放在 join 的左边, 小表放在 join 右边。

15.6 关闭 distributed hash join

若数据存在数据倾斜, 那么 hash join 的性能就会急剧下降。若表足够小以至于可以存储在一个节点的内存中 (通常这种表小于 2GB), 就可以将小表放到右边, 然后用户通过客户端内置的 session 参数将 distributed hash join 关闭。当将 distributed hash join 关闭之后, 两个表均不会在进行 Hash 重分布, 会将右表广播到针对于左表 Source Stage 的每个节点上进行 Join 操作。如下:

```
set session distributed_join = 'false'
SELECT ... FROM large_table l join small_table s on l.join_key = s.join_key
```

15.7 使用 ORC 存储

由于 ORC 是列式结构化存储, 并且当数据量大时, ORC 文件的存储空间比 LZ0 和 Text 文件所使用的存储空间都要小。除此之外, Presto 对于 ORC 文件的读取也做了特定的优化, 因此强烈建议在 Hive 中创建 Presto 使用的表时, 采用 ORC 格式存储。建表语句如下:

```
CREATE TABLE `table1` (
  `id` bigint,
  `name` string,
  `city` string
)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.orc.OrcSerde'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'
```

采用 ORC 格式存储有以下优点:

- ORC 列式存储, 因此查询语句在读取数据时可以避免读取一行中不会被使用的列中的内容, 减少无用数据的读取。
- ORC 是结构化存储文件, 在文件头中存储了很多文件数据的元数据信息, 因此执行统计性查询的时候, 其性能远远高于其他存储格式, 例如执行 Count 查询, 你会发现 ORC 格式的表的查询性能会非常高。
- ORC 文件的存储空间非常小, 尤其随着单个文件存储的数据量越来越大, 相比其他存储格式的文件, ORC 的有效存储率会越来越高。

15.8 小结

本章对 Presto 在使用过程中遇到的性能问题、常用的优化措施及技巧进行了详细说明，通过本章，读者可以快速掌握 Presto 的常用优化技巧，并应用于实际的 Presto 集群的运维和优化工作中，从而使 Presto 集群发挥最高的性能。

第 16 章

Presto 应用场景

从前面的章节我们已经知道，Presto 是一种基于内存的分布式实时计算框架（请注意：实时的概念是指单次执行时间在 200ms 到 20min 之间的查询）。Presto 对于大数据量的实时或者准实时计算具有卓越的性能。我们对 Presto 和 Hive 的性能做了大量的对比测试，最终的结果表明：Presto 的平均查询性能是 Hive 的 10 倍。由于 Presto 的数据源具有完全解耦、高性能，以及对 ANSI SQL 的支持等特性，使得 Presto 在 ETL、实时数据计算、Ad-Hoc 查询和实时数据流分析等多个业务场景中均能发挥重要的作用。

16.1 ETL

由于 Presto 可以方便地支持多种数据源（即使不支持也可以通过实现 Presto 的 SPI 接口方便地实现对新数据源的支持）并且可以支持多种数据源的混合计算，因此我们可以使用简单的 SQL 语句将一个数据源中的数据导入到另一个数据源中。目前官方 Presto 版本仅支持将 MySQL 或者 PostgreSQL 两种 RDBMS，你可以通过 Create Table...As Select...语句将 MySQL 或者 Postgre 中的数据导入到 Hive 中，而且不支持将 Hive 中的数据反向导入到 RDBMS 中。不过我们已经通过二次开发实现了将 Hive 中的数据导入到 MySQL 中。

在某公司现实使用场景中，通过与 azkaban 调度系统配合使用，可以实现定时或者以固定频率调度特定的 Presto 执行脚本（形如 Insert Overwrite Table Partition(...) As Select...的 SQL 语句，目前官方 Presto 版本尚未提供该功能，但是若使用 JD-Presto 则可以提供该功能）来完成相关的 ETL 工作。Presto 的这种使用方式提供了一种全新的 ETL 方案，比传统的 ETL 方案更方便、快捷。如图 16-1 所示是某公司这种应用场景的实际系统架构。

15.8 小结

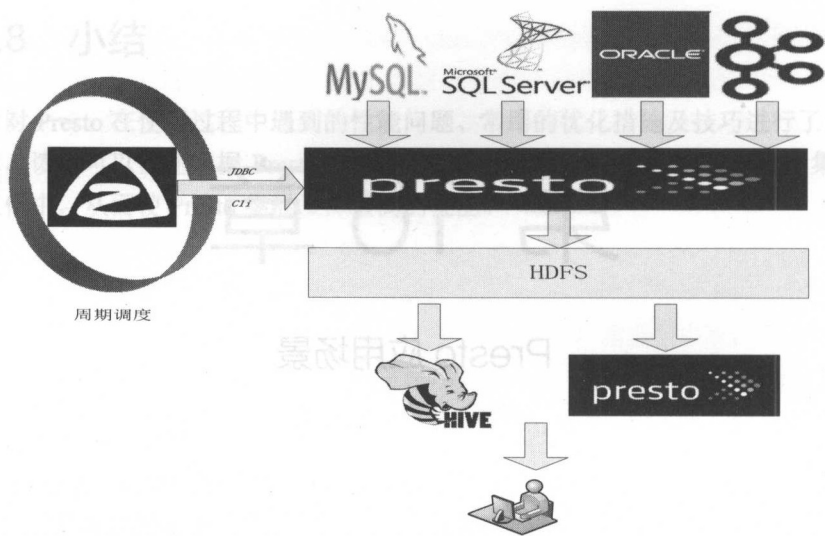


图 16-1

16.2 实时数据计算

由于 Presto 卓越的性能表现，使得 Presto 可以弥补 Hive 无法满足的实时计算空白，因此可以将 Presto 与 Hive 配合使用：对于海量数据的批处理和计算由 Hive 来完成；对于大量数据（单次计算扫描数据量级在 GB 到 TB）的计算由 Presto 完成。Presto 能够完成的实时计算实际上分为以下两种情况。

1. 快照数据实时计算

在这种情况下，可以基于某个时间点的快照数据进行计算，但是要求计算过程快速完成（200ms~20min）。

2. 完全实时计算

要完成完全实时计算，需要满足以下两个条件。

- (1) 使用的基准数据要实时更新，时刻保持与线上实际数据库中的数据完全一致。
- (2) 计算过程要能够快速完成。

在某公司的实际使用场景中，Presto 被用于下述两种业务场景中。

- 基于 $T+1$ 数据的实时计算

在这种业务场景中，用户并不要求基准数据的实时更新，但是要求每次查询数据都能够快速响应。需要 Presto 和 Hive 配合使用来满足实际的业务需求。每天凌晨通过 azkaban 调度 Hive 脚本，根据前一天的数据计算生成中间结果表，生成完毕之后使用 Presto 查询中间结果表，得出用户最终所需要的数据。满足该业务场景的解决方案如图 16-2 所示。

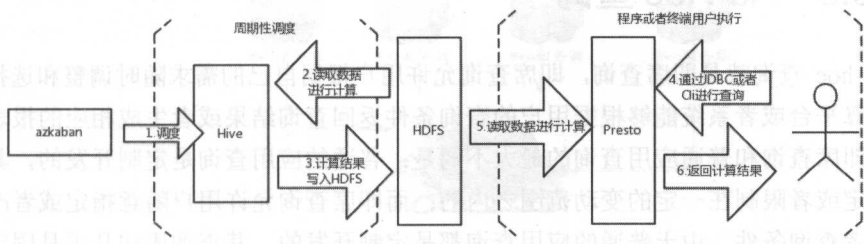


图 16-2

• 基于 RDBMS 的实时计算

在这种业务场景中，用户要求查询的数据完全实时，即只要业务库中的数据发生改变，通过 Presto 查询的时候，就可以查询到刚刚改变之后的数据。要达到这个效果，我们需要使用合理的机制保证数据实时同步，因此我们使用数据库复制技术，为线上的业务数据库建立实时同步的从库，然后用 Presto 查询数据库中的数据，进而进行计算（请注意：使用官方的 Presto 直接读取数据库的性能还太低，因此建议使用 JD-Presto 中的 PDBO 从数据库中读取数据并进行计算）。满足该业务场景的解决方案如图 16-3 所示。

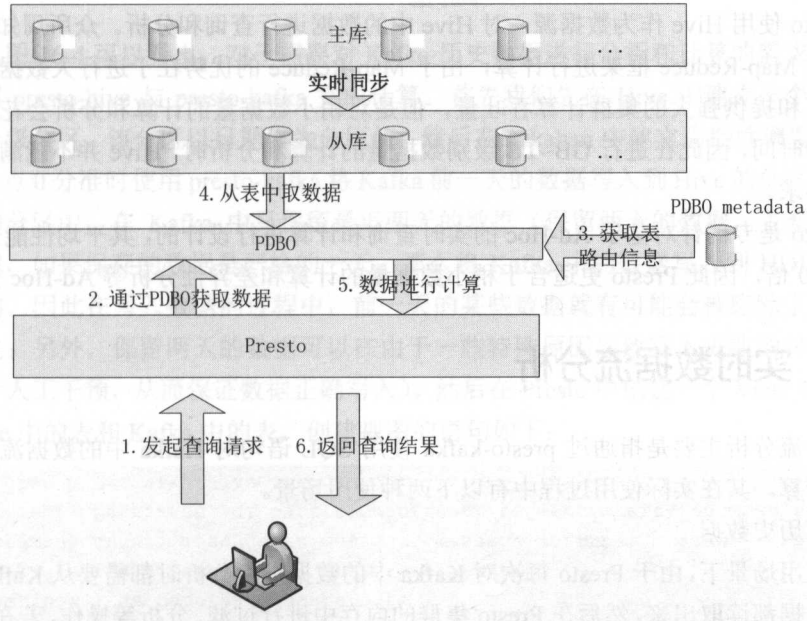


图 16-3

16.3 Ad-Hoc 查询

Ad-hoc 查询就是即席查询，即席查询允许用户根据自己的需求随时调整 and 选择查询条件，计算平台或者系统能够根据用户的查询条件返回查询结果或者生成相应的报表。由此可见，即席查询和普通应用查询的最大不同是：普通的应用查询是定制开发的，其查询语句是固定或者限制在一定的变动范围之内的；而即席查询允许用户随意指定或者改变查询语句或者查询条件。由于普通的应用查询都是定制开发的，其查询语句几乎是固定的，因此，在系统实施时就可以通过建立索引或者分区等技术来优化这些查询，从而提高查询效率。但是即席查询是用户在使用时临时产生的、系统无法预知的，因此也无法对这些查询进行有针对性的优化和改进。

某公司使用 Presto 完成 Ad-Hoc 查询，实际的 Ad-Hoc 使用场景包括以下两种。

(1) 使用 BI 工具进行报表展现

BI 工具通过 ODBC 驱动连接至 Presto 集群，BI 工程师使用 BI 工具进行不同维度的报表设计和展现。由于目前 Facebook 提供的 ODBC 驱动是使用 D 语言开发的，而且功能尚不完善，因此采用 Treasure Data 提供的基于 Presto-gres 中的 ODBC 驱动改造之后的 ODBC 驱动连接到 Presto 集群。

(2) 使用 Cli 客户端进行数据分析

Presto 使用 Hive 作为数据源，对 Hive 中的数据进行查询和分析。众所周知，Hive 使用 Map-Reduce 框架进行计算，由于 Map-Reduce 的优势在于进行大数据量的批运算和提供强大的集群计算吞吐量，但是对稍小数据量的计算和分析会花费相当长的时间，因此在进行 GB~TB 级别数据量的计算和分析时，Hive 并不能满足实时性要求。

Presto 是专门针对基于 Ad-Hoc 的实时查询和计算进行设计的，其平均性能是 Hive 的 10 倍，因此 Presto 更适合于稍小数据量的计算和差异性分析等 Ad-Hoc 查询。

16.4 实时数据流分析

实时数据流分析主要是指通过 presto-kafka 使用 SQL 语句对 Kafka 中的数据流进行清洗、分析和计算。其在实际使用过程中有以下两种使用场景。

(1) 保留历史数据

在这种使用场景下，由于 Presto 每次对 Kafka 中的数据进行分析时都需要从 Kafka 集群中将所有的数据都读取出来，然后在 Presto 集群的内存中进行过滤、分析等操作，若在 Kafka 中保留了大量的历史数据，那么通过 presto-kafka 使用 SQL 语句对 Kafka 中的数据进行分析就会在数据传输上花费大量的时间，从而导致查询效率的降低。因此我们应该避免在 Kafka 中存储大量的数据，从而提高查询性能。

某公司在这种使用场景下，通过使用 presto-hive 与 presto-kafka 配合，完成历史数据的分析和查询。具体系统架构如图 16-4 所示。

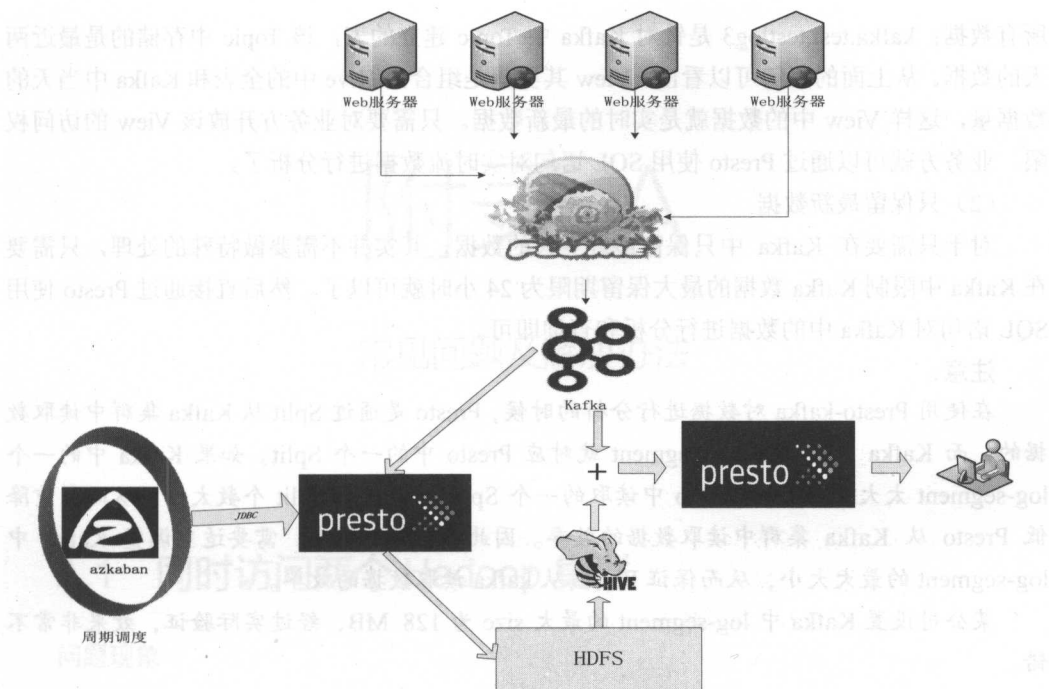


图 16-4

从图 16-4 可以看出，对于需要对 Kafka 历史数据进行分析 and 计算的需求，我们需要配合使用 presto-hive 与 presto-kafka 完成计算。首先我们先在 Hive 中建立一个分区表，分区表有一级分区，该分区以日期作为分区值。然后在 azkaban 中建立周期性调度任务，在每天凌晨 0 点 0 分准时使用 presto-kafka 将 Kafka 前一天的数据写入到 Hive 的分区表前一天日期对应的分区中，在 Kafka 中只保留最近两天的数据（保留两天的数据，主要是出于容错性的考虑，如果保存的数据是严格的一天，那么将 Kafka 中的数据写入到 HDFS 是需要一定时间的，因此在写入数据的过程中，前一天的某些数据就有可能被删除了，从而导致数据丢失；另外，保留两天的数据可以在由于一些特殊原因导致写入失败的时候有充足的时间进行人工干预，从而保证数据正确写入）。然后在 Presto 中创建一个 View（视图）用于组合 Hive 中的表和 Kafka 中的表，创建视图的语句如下：

```
create view kafka_hive_view AS
select _partition_id, _partition_offset, _segment_start, _segment_end,
_segment_count, message as message, _message_corrupt, _message_length,
CAST(_key AS timestamp) as message_timestamp,
cast(now() as date) as message_date
from
kafka.test.testlog3
where _key LIKE concat(substr(cast(now() as varchar),1,10) , '%')
UNION ALL
select * from hive.test.testkafka;
```

在上面的语句中，hive.test.testkafka 是 Hive 中的表，该表用于存储 Kafka 中今天之前的

所有数据；`kafka.test.testlog3` 是针对 Kafka 中 Topic 建立的表，该 Topic 中存储的是最近两天的数据。从上面的语句可以看出：View 其实就是组合的 Hive 中的全表和 Kafka 中当天的数据量，这样 View 中的数据就是实时的最新数据。只需要对业务方开放该 View 的访问权限，业务方就可以通过 Presto 使用 SQL 语句对实时流数据进行分析了。

(2) 只保留最新数据

对于只需要在 Kafka 中只保留最近一天的数据，其实并不需要做特殊的处理，只需要在 Kafka 中限制 Kafka 数据的最大保留期限为 24 小时就可以了。然后直接通过 Presto 使用 SQL 语句对 Kafka 中的数据进行分析和查询即可。

注意：

在使用 Presto-kafka 对数据进行分析的时候，Presto 是通过 Split 从 Kafka 集群中读取数据的，而 Kafka 中的一个 log-segment 就对应 Presto 中的一个 Split。如果 Kafka 中的一个 log-segment 太大就会导致 Presto 中读取的一个 Split 太大，而 Split 个数太少，从而严重降低 Presto 从 Kafka 集群中读取数据的效率。因此在实际应用中，需要适当调整 Kafka 中 log-segment 的最大大小，从而保证 Presto 从 kafka 读取数据的效率。

某公司设置 Kafka 中 log-segment 的最大 size 为 128 MB，经过实际验证，效果非常不错。

16.5 小结

本章对 Presto 适用的所有应用场景进行了详细说明，并对在各个使用场景使用 Presto 的过程中需要注意的事项和有用的技巧进行了讲解。通过本章，读者可以清楚地了解到 Presto 可以应用于哪些实际的应用场景中，并且可以提前了解到在使用过程中需要注意的一些问题，为成功应用 Presto 做准备。

附录 A

常见问题及解决办法

A.1 同时访问两个 Hadoop 集群

问题现象

在 Presto 集群中需要同时访问两个 Hadoop 集群中的 Hive 数据，在 Presto 中针对这两个集群中的 Hive 依次创建了对应的配置文件 `hive1.properties` 和 `hive2.properties`，即分别创建了对应的 Catalog: `hive1` 和 `hive2`。

两个配置文件的内容分别如下：

hive1.properties:

```
connector.name=hive-cdh5
hive.metastore.uri=thrift://hive1-MetaStore:9083
hive.config.resources=/software/servers/confs/hadoop/cluster1/core-site.xml,
/software/servers/confs/hadoop/cluster1/hdfs-site.xml,
/software/servers/confs/hadoop/cluster1/mapred-site.xml
hive.allow-drop-table=true
hive.allow-rename-table=true
```

hive2.properties:

```
connector.name=hive-cdh5
hive.metastore.uri=thrift://hive2-MetaStore:9083
hive.config.resources=/software/servers/confs/hadoop/cluster2/core-site.xml,
/software/servers/confs/hadoop/cluster2/hdfs-site.xml,
/software/servers/confs/hadoop/cluster2/mapred-site.xml
hive.allow-drop-table=true
hive.allow-rename-table=true
```

在 Presto 集群中分别查询这两个 Catalog 中的数据，发现不能同时查询两个 Catalog 中的数据，每次都只能查询其中一个 Catalog 中的数据，而且第一次查询哪个 Catalog 中的数据，那么该 Catalog 中的数据是可查询的，另一个 Catalog 中的数据则是不能查询的。

现象如下:

第一步: 首先查询 hive1 中的数据, 再查询 hive2 中的数据。

```
presto:runtime> select * from hive1.mydb.t1;
  c1 | c2
-----+-----
  aaa | bbb
(1 row)
presto: runtime > select * from hive2.mydb.t1;
Query 20150624_021451_00003_c3t33 failed: Partition location does not exist:
hdfs://ns1/user/hdfs/mydb.db/t1
com.facebook.presto.spi.PrestoException: Partition location does not exist:
hdfs://ns1/user/ hdfs /mydb.db/t1
presto: runtime >
```

第二步: 重启 Presto 集群。

第三步: 首先查询 hive2 中的数据, 再查询 hive1 中的数据。

```
presto: runtime > select * from hive2.mydb.t1;
  id | name
-----+-----
   1 | q1
   2 | q2
   3 | q3
   4 | q4
   5 | q5
   6 | q6
   7 | q7
(7 rows)
presto:app> select * from hive1. mydb.t1;
Query 20150624_022105_00003_ckba7, FAILED, 1 node
Query 20150624_022105_00003_ckba7 failed: Partition location does not exist:
hdfs://ns1/user/hdfs/mydb.db/t1
com.facebook.presto.spi.PrestoException: Partition location does not exist:
hdfs://ns1/user/hdfs/mydb.db/t1
```

问题原因

在获得 FileSystem 的时候是通过方法.FileSystem.get(URI uri, Configuration conf)获得对应的 FileSystem 的, 而该方法最终调用的方法是 CACHE.get(uri, conf)。通过阅读 Hadoop 的源码可以发现, Cache 的 Key 值是根据 URI 生成的, 而这里的 URI 其实就是根据 Hadoop 的配置文件 core-site.xml 中的配置项 fs.defaultFS 生成的, 由于两个 Hadoop 集群的 fs.defaultFS 取的名字一样, 因此, 只能得到其中的一个 FileSystem。

解决方案

选取一个 Hadoop 集群, 将 defaultFs 由 ns1 改为 ns2。操作步骤为:

- (1) 选择需要更改 defaultFs 的集群, 然后关闭 hdfs (在 namenode 上执行 stop-dfs.sh)。
- (2) 在 Namenode 上修改配置文件 core-site.xml 中的配置项如下:

```
<property>
```



```

    <name>fs.defaultFS</name>
    <value>hdfs://ns2</value>
  </property>

```

(3) 在 Namenode 上修改配置文件 `hdfs-site.xml` 中的配置项如下:

```

  <property>
    <name>dfs.nameservices</name>
    <value>ns2</value>
  </property>
  <property>
    <name>dfs.ha.namenodes.ns2</name>
    <value>nn1,nn2</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.ns2.nn1</name>
    <value>namenode1:8020</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.ns2.nn1</name>
    <value>namenode1:50070</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.ns2.nn2</name>
    <value>namenode2:8020</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.ns2.nn2</name>
    <value>namenode2:50070</value>
  </property>
  <property>
    <name>dfs.client.failover.proxy.provider.ns2</name>
    <value>org.apache.hadoop.hdfs.server.namenode.ha.
ConfiguredFailoverProxyProvider</value>
  </property>
  <property>
    <name>dfs.namenode.shared.edits.dir</name>
    <value>qjournal:// qjournal1:8485; qjournal2:8485;
qjournal3:8485/ns2</value>
  </property>

```

(4) 将修改完成配置文件同步到集群的所有节点。

(5) 将 Hadoop 集群进行格式化。

(6) 启动集群。

结果验证

按照上述解决方案操作完成后, 重新执行查询, 验证如下。

第一步: 首先查询 `hive1` 中的数据, 再查询 `hive2` 中的数据。

```

presto:runtime> select * from hive1.mydb.t1;
c1 | c2

```

```

-----+-----
aaa | bbb
(1 row)
presto: runtime > select * from hive2.mydb.t1;
id | name
-----+-----
1 | q1
2 | q2
3 | q3
4 | q4
5 | q5
6 | q6
7 | q7
(7 rows)

```

第二步：首先查询 hive2 中的数据，再查询 hive1 中的数据。

```

presto: runtime > select * from hive2.mydb.t1;
id | name
-----+-----
1 | q1
2 | q2
3 | q3
4 | q4
5 | q5
6 | q6
7 | q7
(7 rows)
presto:runtime> select * from hive1.mydb.t1;
c1 | c2
-----+-----
aaa | bbb
(1 row)

```

由此可见：问题解决，集群恢复正常。

A.2 Kafka 集群重启后无法获取数据

问题现象

首先，在 Kafka 集群中创建两个 Topic: jd.test 和 jd.test2。

```

$ bin/kafka-topics.sh --create --zookeeper 192.168.100.100:2181
--replication-factor 1 --partition 1 --topic jd.test
$ bin/kafka-topics.sh --create --zookeeper 192.168.100.100:2181
--replication-factor 3 --partition 1 --topic jd.test2

```

其中，192.168.100.100 和 2181 分别是部署的 zookeeper 集群中某个节点的 IP 地址和 clientPort。jd.test 设置为 1 个分区，1 个副本；jd.test2 设置为 1 个分区，3 个副本。通过执行 “describe topics” 命令来查看上述两个 Topic 信息：

```

$ bin/kafka-topics.sh --describe --zookeeper 192.168.100.100:2181 --topic jd.test

```

```
Topic:jd.test PartitionCount:1 ReplicationFactor:1 Configs:
Topic: jd.test Partition: 0 Leader: 92 Replicas: 92 Isr: 92

$ bin/kafka-topics.sh --describe --zookeeper 192.168.100.100:2181 --topic
jd.test2
Topic:jd.test2 PartitionCount:1 ReplicationFactor:3 Configs:
Topic: jd.test2 Partition: 0 Leader: 93 Replicas: 93,91,92 Isr:
93,91,92
```

其中，Kafka 中每个 partition 都有一个节点（server）作为该 partition 的 leader，leader 负责该 partition 的所有读写操作。Isr(in-sync replicas)是保持同步的副本所在的节点序列，当 leader 失效的时候，就会在 Isr 中选取新的 leader。现在，我们分别向两个 Topic 中发送若干条数据：

```
$ bin/kafka-console-producer.sh --broker-list 192.168.100.112:9092 --topic
jd.test
Hello, World!
Hello, Presto!
Hello, JD!
$ bin/kafka-console-producer.sh --broker-list 192.168.100.112:9092 --topic
jd.test2
World
Presto
JD
```

在 Kafka 连接器的配置文件中加入上述的两个 Topic，然后重启 Presto 集群，打开 Presto 客户端，对上述两个 Topic 对应的两张表分别执行相同的查询操作：

```
presto:jd> select _partition_offset, _message from test;
_partition_offset | _message
-----+-----
0 | Hello, World!
1 | Hello, Presto!
2 | Hello, JD!

(3 rows)

Query 20150831_095257_00010_cva9y, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:01 [3 rows, 37B] [5 rows/s, 70B/s]
presto:jd> select _partition_offset, _message from test2;
_partition_offset | _message
-----+-----
0 | World
1 | Presto
2 | JD

(3 rows)

Query 20150831_095309_00011_cva9y, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:01 [3 rows, 13B] [4 rows/s, 18B/s]
```


我们发现，发送到两个 Topic 中的消息都被查询出来了。然后，我们重启 Kafka 集群，Kafka 集群重启之后，我们再次执行刚才的查询操作。

```
presto:jd> select _partition_offset, _message from test;
_partition_offset | _message
-----+-----
                0 | Hello, World!
                1 | Hello, Presto!
                2 | Hello, JD!

(3 rows)

Query 20150901_024342_00021_cva9y, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:00 [3 rows, 37B] [79 rows/s, 980B/s]

presto:jd> select _partition_offset, _message from test2;

Query 20150901_024347_00022_cva9y, FAILED, 2 nodes
Splits: 2 total, 0 done (0.00%)
0:00 [0 rows, 0B] [0 rows/s, 0B/s]

Query 20150901_024347_00022_cva9y failed: could not fetch data from Kafka, error
code is '6'
```

我们发现，表 test 的查询结果与之前的查询结果完全相同，而表 test2 的查询出现错误，无法从 Kafka 集群获取数据。

问题原因

再次执行“describe topics”命令来查看上述两个 Topic 信息：

```
$ bin/kafka-topics.sh --describe --zookeeper 192.168.100.100:2181 --topic jd.test
Topic:jd.test PartitionCount:1 ReplicationFactor:1 Configs:
Topic: jd.test Partition: 0 Leader: 92 Replicas: 92 Isr: 92

$ bin/kafka-topics.sh --describe --zookeeper 192.168.100.100:2181 --topic
jd.test2
Topic:jd.test2 PartitionCount:1 ReplicationFactor:3 Configs:
Topic: jd.test2 Partition: 0 Leader: 93 Replicas: 93,91,92 Isr:
92,93,91
```

我们发现，jd.test2 的信息在 kafka 集群重启之后发生了变化。我们查看 Presto 集群的日志文件 server.log 可以发现报错信息如下：

```
2015-09-01T10:43:47.279+0800 DEBUG query-execution-81
com.facebook.presto.execution.QueryStateMachine Query
20150901_024347_00022_cva9y is FAILED
2015-09-01T10:43:47.279+0800 ERROR query-execution-79
com.facebook.presto.execution.QueryStateMachine Query
20150901_024347_00022_cva9y failed
com.facebook.presto.spi.PrestoException: could not fetch data from Kafka, error
code is '6'
```



```

        at
com.facebook.presto.kafka.KafkaRecordSet$KafkaRecordCursor.openFetchRequest(KafkaRecordSet.java:319)
        at
com.facebook.presto.kafka.KafkaRecordSet$KafkaRecordCursor.advanceNextPosition(KafkaRecordSet.java:162)
        at
com.facebook.presto.spi.RecordPageSource.getNextPage(RecordPageSource.java:93)
        at
com.facebook.presto.operator.TableScanOperator.getOutput(TableScanOperator.java:239)
        at com.facebook.presto.operator.Driver.processInternal(Driver.java:380)
        at com.facebook.presto.operator.Driver.processFor(Driver.java:303)
        at
com.facebook.presto.execution.SqlTaskExecution$DriverSplitRunner.processFor(SqlTaskExecution.java:587)
        at
com.facebook.presto.execution.TaskExecutor$PrioritizedSplitRunner.process(TaskExecutor.java:505)
        at
com.facebook.presto.execution.TaskExecutor$Runner.run(TaskExecutor.java:640)
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)

```

可以看到异常抛出的位置在 `KafkaRecordSet` 类的 319 行，通过查看源代码，问题出现在 313 行：

```
SimpleConsumer consumer = consumerManager.getConsumer(split.getNodes().get(0));
```

其中 `Split` 是 `KafkaSplit` 的实例，`KafkaSplit` 是在 `KafkaSplitManager` 类中的 `getPartitionSplits()` 方法中创建的，代码如下：

```

KafkaSplit split = new KafkaSplit(connectorId,
    partition.getTopicName(),
    kafkaTableHandle.getKeyDataFormat(),
    kafkaTableHandle.getMessageDataFormat(),
    partition.getPartitionIdAsInt(),
    offsets[i],
    offsets[i - 1],
    partition.getPartitionNodes());

```

可以看出 `split.getNodes().get(0)` 是获取的 `partition.getPartitionNodes()` 的第一个节点，而 `partition` 实例是在 `KafkaSplitManager` 类中的 `getPartitions()` 方法中创建的，代码如下：

```

builder.add(new KafkaPartition(metadata.topic(),
    part.partitionId(),
    HostAddress.fromParts(leader.host(), leader.port()),
    ImmutableList.copyOf(Lists.transform(part.isr(),
KafkaSplitManager::brokerToHostAddress))));

```

可以看到, `split.getNode().get(0)` 获取的 `HostAddress` 实例是 `Isr` 中的第一个节点, 这里是 92 节点, 而该 `Split` 所在的分区的 `leader` 是 93 节点。只有 `Split` 所在的 `Kafka` 分区对应的 `leader` 创建的 `consumer` 才能对该分区进行读写操作, 显然 92 节点对该分区的数据没有读写操作的权限, 所以当进行查询的时候会报错, 无法从 `Kafka` 集群获取数据。而对于像 `jd.test` 这样只有一个备份的 `Topic`, 获取到的始终是 `leader` 节点, 所以在 `Presto` 中进行查询不会出现错误。

解决方案

找到出错的原因后, 我们应该让 `KafkaPartition` 实例中的属性 `partitionNodes` 中第一个位置放置该 `Partition` 的 `Leader` 节点对应的 `HostAddress` 实例即可。这样, 对 `KafkaSplitManager` 类中的 `getPartitions()` 方法中的 `if...else` 语句中的 `else` 部分更改如下:

```
else {
    ImmutableList.Builder<HostAddress> partitionNodesBuilder =
ImmutableList.builder();
    HostAddress partitionLeader = brokerToHostAddress(leader);

    boolean isLeaderFirst = true;
    //首先判断第一个 isr 是不是 leader
    if (part.isr().get(0).id() != leader.id()) {
        isLeaderFirst = false;
    }
    //若第一个 isr 不是 leader 就优先将 leader 放入到 builder 中
    if (!isLeaderFirst) {
        partitionNodesBuilder.add(partitionLeader);
    }
    //遍历所有的 isr, 将剩下的 isr 依次放入到 builder 中
    for (Broker broker : part.isr()) {
        if (isLeaderFirst || broker.id() != leader.id()) {
            partitionNodesBuilder.add(brokerToHostAddress
(broker));
        }
    }

    builder.add(new KafkaPartition(metadata.topic(),
        part.partitionId(),
        partitionLeader,
        partitionNodesBuilder.build()));
}
```

通过上面的代码就可以保证每次取到的 `isr` 都是 `leader`, 这样就可以保证 `split.getNodes().get(0)` 获取的是 `split` 所在 `Partition` 的 `Leader` 节点对应的 `HostAddress` 实例。

结果验证

按照上述解决方案操作完成后, 将更改后的代码重新进行编译、部署和运行, 重启 `Kafka` 集群, 再次执行 “`describe topics`” 命令来查看 `jd.test2` 信息:

```
$ bin/kafka-topics.sh --describe --zookeeper 192.168.100.100:2181 --topic
jd.test2
```

```
Topic:jd.test2 PartitionCount:1 ReplicationFactor:3 Configs:
Topic: jd.test2 Partition: 0 Leader: 93 Replicas: 93,91,92 Isr:
92,93,91
```

Isr 中第一个节点仍然不是 Leader, 然后我们打开 Presto 客户端, 再次对 test2 进行查询:

```
presto:jd> select _partition_offset, _message from test2;
_partition_offset | _message
-----+-----
0 | World
1 | Presto
2 | JD
(3 rows)

Query 20150902_033800_00007_cva9y, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:00 [3 rows, 13B] [43 rows/s, 189B/s]
```

可见, 问题已经得到解决。

A.3 Task exceeded max memory size

问题现象

在 Presto 中执行查询语句时出错如下:

```
presto:orc> select count(*) from (select distinct l_partkey from lineitem);

Query 20140904_020831_00017_7u3gw, FAILED, 14 nodes
Splits: 251 total, 0 done (0.00%)
0:02 [55.3M rows, 1.39GB] [34.1M rows/s, 879MB/s]

Query 20140904_020831_00017_7u3gw failed: Task exceeded max memory size of 2MB
presto:orc>
```

查看日志文件, 存在错误提示及堆栈如下:

```
Query 20140904_020831_00017_7u3gw failed: Task exceeded max memory size of 2MB
com.facebook.presto.operator.OperatorContext.setMemoryReservation(OperatorCon
text.java:224)
com.facebook.presto.operator.PagesIndex.addPage(PagesIndex.java:122)
com.facebook.presto.operator.HashBuilderOperator.addInput(HashBuilderOperator
.java:159)
com.facebook.presto.operator.Driver.process(Driver.java:336)
com.facebook.presto.operator.Driver.processFor(Driver.java:272)
com.facebook.presto.execution.SqlTaskExecution$DriverSplitRunner.processFor(S
qlTaskExecution.java:557)
com.facebook.presto.execution.TaskExecutor$PrioritizedSplitRunner.process(Tas
kExecutor.java:444)
com.facebook.presto.execution.TaskExecutor$Runner.run(TaskExecutor.java:578)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:114
5)
```



```
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
```

问题原因

Presto 会将每个查询语句最终分解为在每个节点上运行的 Task, 并且会在配置文件中配置每个 Task 可以使用的最大内存大小。若某个查询语句中的某个 Task 处理的数据超过给 Task 可以使用的最大内存大小, 则会抛出上面的错误。

解决方案

修改 Presto 配置文件 config.properties, 提高其中的配置属性 task.max-memory 的值。

结果验证

```
presto:orc> select count(*) from (select distinct l_partkey from lineitem);
 _col0
-----
20000000
(1 row)

Query 20150916_095332_00003_iaj7e, FINISHED, 14 nodes
Splits: 251 total, 251 done (100.00%)
0:07 [600M rows, 15.1GB] [82.4M rows/s, 2.07GB/s]
```

A.4 SQL 中 In 子句太长导致栈溢出错误

问题现象

在执行一个带有 in 过滤条件的 SQL 语句时, 在 in 条件中包含有大量的值。Presto 查询失败, 然后查看 Coordinator 中的日志, 可以看到如下错误信息:

```
RuntimeException: statement is too large (stack overflow during analysis)
  at com.facebook.presto.execution.SqlQueryExecution.analyzeQuery
(SqlQueryExecution.java:194)
  at com.facebook.presto.execution.SqlQueryExecution.start
(SqlQueryExecution.java:151)
  at com.facebook.presto.execution.QueuedExecution.lambda$start$67
(QueuedExecution.java:68)
  at com.facebook.presto.execution.QueuedExecution$$Lambda$143/696860887.
run(Unknown Source)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1142)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:617)
  at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.StackOverflowError
  at com.facebook.presto.sql.tree.AstVisitor.process(AstVisitor.java:22)
  at com.facebook.presto.sql.planner.DomainTranslator$Visitor.
visitLogicalBinaryExpression(DomainTranslator.java:255)
  at com.facebook.presto.sql.planner.DomainTranslator$Visitor.
```



```

visitLogicalBinaryExpression(DomainTranslator.java:200)
    at com.facebook.presto.sql.tree.LogicalBinaryExpression.accept
(LogicalBinaryExpression.java:59)
    at com.facebook.presto.sql.tree.AstVisitor.process(AstVisitor.java:22)
    at
com.facebook.presto.sql.planner.DomainTranslator$Visitor.visitLogicalBinaryExpress
ion(DomainTranslator.java:255)
    at com.facebook.presto.sql.planner.DomainTranslator$Visitor.
visitLogicalBinaryExpression(DomainTranslator.java:200)
    at com.facebook.presto.sql.tree.LogicalBinaryExpression.accept
(LogicalBinaryExpression.java:59)
    at com.facebook.presto.sql.tree.AstVisitor.process(AstVisitor.java:22)
    at com.facebook.presto.sql.planner.DomainTranslator$Visitor.
visitLogicalBinaryExpression(DomainTranslator.java:255)
    at com.facebook.presto.sql.planner.DomainTranslator$Visitor.
visitLogicalBinaryExpression(DomainTranslator.java:200)
    at com.facebook.presto.sql.tree.LogicalBinaryExpression.accept
(LogicalBinaryExpression.java:59)
    at com.facebook.presto.sql.tree.AstVisitor.process(AstVisitor.java:22)

```

问题原因

(1) JDK1.8 默认地为每个线程分配的堆栈空间的大小为 1MB。

(2) Presto 解析器在解析 in 语句时有递归调用，当 in 语句太长时，递归的深度太深会导致堆栈溢出错误。

解决方案

在 Presto 的 JVM 配置文件中，通过修改 Xss 的值增加堆栈的大小，配置如下：

```

-server
-Xss8M
-Xmx120G
-XX:+UseConcMarkSweepGC
-XX:+ExplicitGCInvokesConcurrent
-XX:+CMSClassUnloadingEnabled
-XX:+AggressiveOpts
-XX:+HeapDumpOnOutOfMemoryError
-XX:OnOutOfMemoryError=kill -9 %p
-XX:ReservedCodeCacheSize=150M

```

结果验证

语句可以正常执行。

A.5 高并发导致大量查询出错

问题现象

在执行 50 个并发的大数据量查询测试的时候，查询出错，错误信息为：

```

encountered too many errors talking to a worker node.

```

```

com.facebook.presto.operator.PageTransportTimeoutException: Encountered too many errors talking to a worker node. The node may have crashed or be under too much load. This is probably a transient issue, so please retry your query in a few minutes. (http://192.168.100.12:8001/v1/task/20150417_071838_00060_qpydm.2.12/results/20150417_071838_00060_qpydm.1.13/14162 - requests failed for 600.08s)
    at com.facebook.presto.operator.HttpPageBufferClient$1.onFailure(HttpPageBufferClient.java:325)
    at com.google.common.util.concurrent.Futures$6.run(Futures.java:1310)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: io.airlift.http.client.RuntimeIOException: Server refused connection: http://192.168.100.112:8100/v1/task/20150417_071838_00060_qpydm.2.12/results/20150417_071838_00060_qpydm.1.13/14162
    at io.airlift.http.client.ResponseHandlerUtils.propagate(ResponseHandlerUtils.java:17)
    at com.facebook.presto.operator.HttpPageBufferClient$PageResponseHandler.handleException(HttpPageBufferClient.java:434)
    at com.facebook.presto.operator.HttpPageBufferClient$PageResponseHandler.handleException(HttpPageBufferClient.java:1)
    at io.airlift.http.client.jetty.JettyHttpClient$JettyResponseFuture.failed(JettyHttpClient.java:766)
    at io.airlift.http.client.jetty.JettyHttpClient$BufferingResponseListener.onComplete(JettyHttpClient.java:997)
    at org.eclipse.jetty.client.ResponseNotifier.notifyComplete(ResponseNotifier.java:193)
    at org.eclipse.jetty.client.ResponseNotifier.notifyComplete(ResponseNotifier.java:185)
    at org.eclipse.jetty.client.HttpExchange.notifyFailureComplete(HttpExchange.java:271)
    at org.eclipse.jetty.client.HttpExchange.abort(HttpExchange.java:242)
    at org.eclipse.jetty.client.HttpConversation.abort(HttpConversation.java:141)
    at org.eclipse.jetty.client.HttpRequest.abort(HttpRequest.java:704)
    at org.eclipse.jetty.client.HttpDestination.abort(HttpDestination.java:252)
    at org.eclipse.jetty.client.PoolingHttpDestination$1.run(PoolingHttpDestination.java:68)
    at org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:635)
    at org.eclipse.jetty.util.thread.QueuedThreadPool$3.run(QueuedThreadPool.java:555)
    ... 1 more
Caused by: java.net.ConnectException: 拒绝连接
    at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
    at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:716)

```

```
at org.eclipse.jetty.io.SelectorManager.finishConnect(SelectorManager.java:340)
```

问题原因

在 Presto 的配置文件: \$PRESTO_HOME/etc/config.properties 中提高了配置参数: task.max-worker-threads=256, 从而导致每个节点上同时处理的 Split 的个数增多, 最终导致集群中节点的内存使用率过高。

解决方案

修改以下 3 个参数:

- (1) 修改 Presto 的配置文件: \$PRESTO_HOME/etc/config.properties, 将配置参数 task.max-worker-threads 的值修改为 128。
- (2) 根据实际情况提高配置项: query.remote-task.max-consecutive-error-count 的值。
- (3) 根据实际情况提高配置项: query.remote-task.min-error-duration 的值。

上述三个参数的含义可以查看【附录 B】。

结果验证

可以在 50 个并发的情况下成功执行同样的大数据量查询。

附录 B

Presto 配置参数说明

参数名称	所属配置文件	说 明
discovery-server.enabled	etc/config.properties	Coordinator 节点是否同时开启 discovery server
node-scheduler.include-coordinator	etc/config.properties	是否在 Coordinator 节点上分配计算任务
node-scheduler.location-aware-scheduling-enabled	etc/config.properties	是否启用数据位置本地化调度，优先选择 Split 所在的 worker 节点
node-scheduler.max-pending-splits-per-node-per-task	etc/config.properties	当节点所分配的 Split 个数达到 node-scheduler.max-splits-per-node 所设定的参数值时，还可分配额外的最大 Split 的个数，判断逻辑为：当节点已分配 Split 个数达到 node-scheduler.max-splits-per-node 最大值时，判断需要分配的 Split 个数和节点正在等待队列里的 Split 个数之和是否小于此参数值，如果小于此值，则仍可选择该节点进行分配
node-scheduler.max-splits-per-node	etc/config.properties	每个计算节点允许分配最大 Split 个数
node-scheduler.multiple-tasks-per-node-enabled	etc/config.properties	调度器选择备选节点（candidateNodes）时，是否允许在同一个 Stage 过程中多次选择同一个节点备选
node-scheduler.min-candidates	etc/config.properties	Source Stage 最小备选节点个数（candidateNodes）
node.binary-spec	etc/node.properties	节点 JVM 二进制规范
node.config-spec	etc/node.properties	节点 JVM 配置
node.environment	etc/node.properties	node 节点环境名称，属于同一个 Presto 集群的每个节点该名称必须一致
node.location	etc/node.properties	节点 JVM 路径
node.bind-ip	etc/node.properties	node 绑定 server socket 时使用的 IP 地址，如果未设定，则绑定时使用任意 IPv4 地址
node.external-address	etc/node.properties	node 外网地址

续表

参数名称	所属配置文件	说 明
node.id	etc/node.properties	节点 ID，要求每个节点 ID 都是唯一的
node.ip	etc/node.properties	节点内网地址
node.pool	etc/node.properties	节点所在 pool 名称
query.client.timeout	etc/config.properties	客户端查询超时时间，如果超过该时间客户端无 heartbeat，则丢弃该查询，并抛出 AbandonedException 异常
query.initial-hash-partitions	etc/config.properties	初始 hash partition 个数，fix stage 使用这个参数决定分配的节点个数
experimental.big-query-initial-hash-partitions	etc/config.properties	启用 big-query 时，初始 hash partition 个数，默认为活跃节点个数
query.max-age	etc/config.properties	SqlQueryManager 缓存了历史查询信息，若内存中保留的 query 个数超过了 query.max-history 所配置的值，则会把超过了 query.max-age 的 query 从内存中移除
query.max-history	etc/config.properties	SqlQueryManager 查询历史缓存最大保留个数
query.manager-executor-pool-size	etc/config.properties	SqlQueryManager 查询管理器线程个数
query.remote-task.max-consecutive-error-count	etc/config.properties	task 允许连续出错的次数。 该参数需要与参数 query.remote-task.min-error-duration 配合使用
query.remote-task.min-error-duration	etc/config.properties	Task 出错判断的最小时间间隔。 该参数需要与参数 query.remote-task.max-consecutive-error-count 配合使用 说明： 若在 query.remote-task.min-error-duration 时间间隔内，task 请求连续失败 query.remote-task.max-consecutive-error-count 次，则该 task 失败
query.schedule-split-batch-size	etc/config.properties	批量调度 Split 的个数
task.client.timeout	etc/config.properties	task heartbeat 超时时间，如果超过此值则丢弃该 task，并抛出 AbandonedException 异常
task.info.max-age	etc/config.properties	SqlTaskManager 缓存 tasks 的过期时间，task 结束时间距离现在时间间隔若超过了此参数的值，则从 SqlTaskManager 内存中移除
task.max-worker-threads	etc/config.properties	其默认值为节点 cores*4，表示单个 Worker 处理 Split 的最大并发度
task.max-memory	etc/config.properties	单个 task 最大可用内存数
experimental.big-query-max-task-memory	etc/config.properties	big query 支持的最大 task 可用内存
task.operator-pre-allocated-memory	etc/config.properties	每个 operator（目前 TopNRowNumberOperator/AggregationOperator/HashAggregationOperator）预分配的内存
sink.max-buffer-size	etc/config.properties	SqlTask outputbuffer 最大值
task.cpu-timer-enabled	etc/config.properties	是否统计线程 cpu 时间，通过 ThreadMXBean 实行

续表

参数名称	所属配置文件	说 明
maven.repo.local	etc/config.properties	plugin maven 本地 repository
maven.repo.remote	etc/config.properties	plugin maven 远程 repository
plugin.config-dir	etc/config.properties	plugin 配置目录
plugin.dir	etc/config.properties	已安装的 plugin 目录
plugin.bundles	etc/config.properties	plugin 列表, “,” 分隔
coordinator	etc/config.properties	是否为 Coordinator 节点
presto.version	etc/config.properties	版本
experimental-syntax-enabled	etc/config.properties	是否支持正在试验的语法 (approximate、Rescaling)
compiler.interpreter-enabled	etc/config.properties	Presto 会将执行计划中的 ScanFilterAndProjectOperator 和 FilterAndProjectOperator 动态编译为 Byte Code, 并交给 JIT 去编译为 native 代码, 该参数控制如果动态编译失败是否执行, (compilation failed, use interpreter), 如果执行则设为 true, 如果动态编译失败, 则查询不会失败, 但会执行较慢
compiler.new-bytecode-generator-enabled	etc/config.properties	是否启用新的字节码编译 (release 0.74 This version includes new infrastructure for bytecode compilation, and lays the groundwork for future improvements.)
service-inventory.uri	etc/config.properties	Discovery client 通过 service inventory API 发现可用的 discovery servers 地址, 此参数为 service inventory 的 URI, 此功能为实验性, 目前仍支持 discovery.uri 方式设置 discovery server 地址, 此参数为 airlift 参数, 经测试, 目前 Presto 还不支持此功能
service-inventory.update-interval	etc/config.properties	Service inventory 服务的更新时间间隔
discovery.uri	etc/config.properties	discovery server 的地址, discovery-server.enabled=true 时和 coordinator 节点的 URI 相同
discovery.http-client.key-store-path	etc/config.properties	discovery announcement client 数字证书路径
discovery.http-client.max-connections	etc/config.properties	未用到
discovery.http-client.max-connections-per-server	etc/config.properties	discovery announcement client 到每个 destination 的最大连接数
discovery.http-client.max-content-length	etc/config.properties	discovery announcement client 读取数据时 input stream 队列最大 buffer 值
discovery.http-client.read-timeout	etc/config.properties	discovery announcement client 读取数据超时时间
discovery.http-client.socks-proxy	etc/config.properties	discovery announcement client proxy 设置
dynamic.store.gc-interval	etc/config.properties	
dynamic.store.remote.max-batch-size	etc/config.properties	

续表

参数名称	所属配置文件	说 明
dynamic.store.remote.queue-size	etc/config.properties	
dynamic.store.remote.update-interval	etc/config.properties	
dynamic.store.remote.replication-interval	etc/config.properties	
dynamic.store.tombstone-max-age	etc/config.properties	
failure-detector.enabled	etc/config.properties	是否启用节点健康度检测
failure-detector.expiration-grace-interval	etc/config.properties	当 discovery server 获取不到某个节点 service 时，节点状态变为 disabled，该参数为从节点 disabled 到确认节点下线的超时时间，确认节点下线状态顺序为 active->disabled->expired，超过这个时间则将该节点从节点列表删除
failure-detector.threshold	etc/config.properties	确认节点连接成功（successTransition）的阈值，当一定时间范围内心跳检测失败次数和请求次数的比率超过该值时，则将 successTransition 置为 null，表示连接失败，节点为失败节点
failure-detector.heartbeat-interval	etc/config.properties	心跳周期
failure-detector.warmup-interval	etc/config.properties	从认为节点连接成功（successTransition）到确认节点存活的时间间隔
failure-detector.http-client.connect-timeout	etc/config.properties	FailureDetectorModule client Discovery server 和 worker http 连接超时时间
failure-detector.http-client.key-store-password	etc/config.properties	FailureDetectorModule client 数字证书密码
failure-detector.http-client.key-store-path	etc/config.properties	FailureDetectorModule client 数字证书路径
failure-detector.http-client.max-connections-per-server	etc/config.properties	FailureDetectorModule client 到每个 destination 的最大连接数
failure-detector.http-client.max-content-length	etc/config.properties	FailureDetectorModule client http input stream 队列最大 buffer 值
failure-detector.http-client.read-timeout	etc/config.properties	FailureDetectorModule client 读取数据超时时间
failure-detector.http-client.socks-proxy	etc/config.properties	FailureDetectorModule client proxy 设置
failure-detector.http-client.max-threads	etc/config.properties	FailureDetectorModule client 线程池最大线程数
failure-detector.http-client.min-threads	etc/config.properties	FailureDetectorModule client 线程池最小线程数

续表

参数名称	所属配置文件	说 明
exchange.client-threads	etc/config.properties	每个 worker 节点都有一个 ExchangeClient 守护进程, 用于从其他 worker 节点获取中间数据, 此参数控制 ExchangeClient 中并发线程池的个数
exchange.concurrent-request-multiplier	etc/config.properties	控制 ExchangeClient 中的 HttpPageBufferClient 个数, Exchange client 需要根据所需数据的 location 创建 HttpPageBufferClient 从其他 stage outputbuffer 请求数据, 并维护着一个 client 队列, 此参数协助控制一次从队列中提取多少个 client 并发送请求 $clientCount = (int) ((1.0 * neededBytes / averageBytesPerRequest) * concurrentRequestMultiplier);$ $clientCount = Math.max(clientCount, 1);$
exchange.max-buffer-size	etc/config.properties	ExchangeClient buffer 最大值
exchange.max-response-size	etc/config.properties	http page buffer client 设置的 presto header “X-Presto-Max-Size” 值
exchange.min-error-duration	etc/config.properties	Exchange 节点间请求数据时超时时间
exchange.http-client.connect-timeout	etc/config.properties	exchange client http client 连接超时时间
exchange.http-client.key-store-password	etc/config.properties	exchange client http client 数字证书密码
exchange.http-client.key-store-path	etc/config.properties	exchange client http client 数字证书路径
exchange.http-client.max-connections-per-server	etc/config.properties	exchange client http client 到每个 destination 的最大连接数
exchange.http-client.max-content-length	etc/config.properties	exchange client http client input stream 队列最大 buffer 值
exchange.http-client.read-timeout	etc/config.properties	exchange client http client 读取数据超时时间
exchange.http-client.socks-proxy	etc/config.properties	exchange client http client proxy 设置
scheduler.http-client.connect-timeout	etc/config.properties	execution scheduler module http client 连接超时时间
scheduler.http-client.key-store-password	etc/config.properties	execution scheduler module http client 数字证书密码
scheduler.http-client.key-store-path	etc/config.properties	execution scheduler module http client 数字证书路径
scheduler.http-client.max-connections-per-server	etc/config.properties	execution scheduler module http client 到每个 destination 的最大连接数
scheduler.http-client.max-content-length	etc/config.properties	execution scheduler module http client input stream 队列最大 buffer 值

续表

参数名称	所属配置文件	说 明
scheduler.http-client.read-timeout	etc/config.properties	execution scheduler module http client 读取数据超时时间
scheduler.http-client.socks-proxy	etc/config.properties	execution scheduler module http client proxy 设置
execute.http-client.connect-timeout	etc/config.properties	execute resource module http client 连接超时时间
execute.http-client.key-store-password	etc/config.properties	execute resource module http client 数字证书密码
execute.http-client.key-store-path	etc/config.properties	execute resource module http client 数字证书路径
execute.http-client.max-connections-per-server	etc/config.properties	execute resource module http client 到每个 destination 的最大连接数
execute.http-client.max-content-length	etc/config.properties	execute resource module http client input stream 队列最大 buffer 值
execute.http-client.read-timeout	etc/config.properties	execute resource module http client 读取数据超时时间
execute.http-client.socks-proxy	etc/config.properties	execute resource module http client proxy 设置
event.http-client.connect-timeout	etc/config.properties	event module http client 连接超时时间
event.http-client.key-store-password	etc/config.properties	event module http client 数字证书密码
event.http-client.key-store-path	etc/config.properties	event module http client 数字证书路径
event.http-client.max-connections-per-server	etc/config.properties	event module http client 到每个 destination 的最大连接数
event.http-client.max-content-length	etc/config.properties	event module http client input stream 队列最大 buffer 值
event.http-client.read-timeout	etc/config.properties	event module http client 读取数据超时时间
event.http-client.socks-proxy	etc/config.properties	event module (query monitor) proxy 设置
http-client.max-threads	etc/config.properties	http client 线程池最大线程个数
http-client.min-threads	etc/config.properties	http client 线程池最小线程个数
http-server.admin.enabled	etc/config.properties	jetty server 是否启用基于 NIO 的 admin connector 并绑定指定端口用于 JMX 管理, 如果不设置, 则随机绑定端口
http-server.admin.threads.max	etc/config.properties	jetty server admin connector 线程池最大线程数
http-server.admin.threads.min	etc/config.properties	jetty server admin connector 线程池最小线程数
http-server.admin.port	etc/config.properties	jetty server admin connector 端口号
http-server.http.enabled	etc/config.properties	work 节点 jetty server 是否启用 http connector
http-server.http.port	etc/config.properties	http server 端口号

续表

参数名称	所属配置文件	说 明
http-server.https.enabled	etc/config.properties	work 节点 jetty serve 是否启用 https connector
http-server.https.port	etc/config.properties	https 端口
http-server.https.keystore.key	etc/config.properties	jetty server 启用 https 时数字证书密码
http-server.https.keystore.path	etc/config.properties	jetty server 启用 https 时数字证书路径
http-server.log.path	etc/config.properties	server log 路径
http-server.log.retention-time	etc/config.properties	server log 最大保留时间
http-server.max-request-header-size	etc/config.properties	一个 Http 请求头的最大值
http-server.threads.max	etc/config.properties	server 线程池最大线程个数
http-server.threads.min	etc/config.properties	server 线程池最小线程个数
http-server.net.max-idle-time	etc/config.properties	server 连接最大空闲（超时）时间
http-server.threads.max-idle-time	etc/config.properties	jetty server 线程池最大空闲时间
http-server.auth.users-file	etc/config.properties	jetty HashLoginService 配置文件
jmx.rmiregistry.port	etc/config.properties	jmx 注册端口
jmx.rmiserver.port	etc/config.properties	jmx 端口
hive.metastore-cache-ttl	etc/config.properties	hive 元数据缓存的过期时间
hive.metastore-refresh-interval	etc/config.properties	hive 元数据缓存的刷新时间
optimizer.optimize-hash-generation	etc/config.properties	表示是否启用对哈希聚合的优化，如果启用该选项，将会提升 CPU 使用率
query.max-queued-queries	etc/config.properties	普通查询队列的最大容量
query.max-concurrent-queries	etc/config.properties	普通查询最大并发度

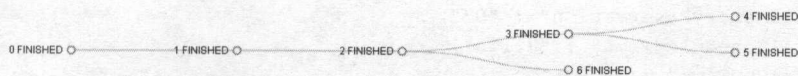
Presto 执行信息说明

在本次查询中，涉及的表有以下几个。

- 查询 SOL 语句如下。

- 第一行中所显示的信息为整个查询的大致简要信息。

Stages



Stage 关系图下面显示的便是当前查询详细的 Tasks 信息，可以查看到 Task 执行的所在节点，当前 Task 运行的 Split 个数等。除此之外，还可以访问 RESTful 所定义的 URI 来查看更加详细的信息，例如访问类似于 `http://Coordinator_ip:port/v1/query/queryID?pretty` 的 URL，便可以 JSON 格式的信息呈现出指定 QueryID 的详细信息，例如访问本例的 URI：`http://Coordinator_ip:port/v1/query/20150914_085716_00019_zgnuq?pretty`，便可以查看本次查询的详细信息，如下所示：

```
"queryId" : "20150914_085716_00019_zgnuq",
"session" : {
  "user" : "hadp",
  "source" : "presto-cli",
  "catalog" : "hive",
  "schema" : "employees",
  "timeZoneKey" : 1980,
  "locale" : "zh_CN",
  "remoteUserAddress" : "192.168.100.10",
  "userAgent" : "StatementClient/0.107-1-g0cf7e34-dirty",
  "startTime" : 1442221036088,
  "systemProperties" : { },
  "catalogProperties" : { }
},
"state" : "FINISHED",
"memoryPool" : "general",
"scheduled" : true,
"self" : "http:// 192.168.100.10:8001/v1/query/20150914_085716_00019_zgnuq",
"fieldNames" : [ "dept_name", "_coll" ],
"query" : "select dept_name,sum(salary) from dept_emp a left join salaries b on
a.emp_no = b.emp_no left join departments c on a.dept_no = c.dept_no group by
c.dept_name",
"queryStats" : {
  "createTime" : "2015-09-14T08:57:16.089Z",
  "executionStartTime" : "2015-09-14T08:57:16.112Z",
  "lastHeartbeat" : "2015-09-14T08:57:18.525Z",
  "endTime" : "2015-09-14T08:57:18.525Z",
  "elapsedTime" : "2.44s",
  "queuedTime" : "601.59us",
  "analysisTime" : "17.70ms",
  "distributedPlanningTime" : "3.08ms",
  "totalPlanningTime" : "21.41ms",
  "totalTasks" : 85,
  "runningTasks" : 0,
  "completedTasks" : 85,
  "totalDrivers" : 137,
```

```

"queuedDrivers" : 0,
"runningDrivers" : 0,
"completedDrivers" : 137,
"totalMemoryReservation" : "0B",
"totalScheduledTime" : "7.11s",
"totalCpuTime" : "6.92s",
"totalUserTime" : "6.60s",
"totalBlockedTime" : "2.99m",
"fullyBlocked" : true,
"blockedReasons" : [ ],
"rawInputDataSize" : "42.08MB",
"rawInputPositions" : 3175659,
"processedInputDataSize" : "81.77MB",
"processedInputPositions" : 3175659,
"outputDataSize" : "261B",
"outputPositions" : 9
},

```

其中 session 信息段显示了当前查询的 session 相关信息、state 状态、fieldNames 输出的字段名称、query 语句等，而 queryStats 信息块中则详细记录了整个查询的执行信息，例如各种时间的统计、task 的统计、数据量的统计等。

上面的内容描述的是整个查询的信息，而下面的内容非常详细地显示出每个 Stage 和 Tasks 的信息，比如 Stage 的逻辑执行计划信息，Task 之间的数据传输信息等：

```

"stageId" : "20150914_085716_00019_zgnuq.1",
"state" : "FINISHED",
"self" : "http:// 192.168.100.10:8001/v1/stage/20150914_085716_00019_zgnuq.1",
"plan" : {
  "id" : "5",
  "root" : {
    "type" : "aggregation",
    "id" : "23",
    "source" : {
      "type" : "remoteSource",
      "id" : "29",
      "sourceFragmentIds" : [ "4" ],
      "outputs" : [ "dept_name", "$hashvalue_20", "sum_21" ]
    },
    "groupBy" : [ "dept_name" ],
    "aggregations" : {
      "sum" : "\"sum\"(\"sum_21\")"
    },
    "functions" : {
      "sum" : {
        "name" : "sum",
        "typeParameters" : [ ],
        "returnType" : "bigint",
        "argumentTypes" : [ "bigint" ],
        "variableArity" : false,
        "internal" : false
      }
    }
  }
}

```

```

    }
  },
  "masks" : { },
  "step" : "FINAL",
  "sampleWeight" : null,
  "confidence" : 1.0,
  "hashSymbol" : "$hashvalue_20"
},
"symbols" : {
  "$hashvalue_20" : "bigint",
  "sum" : "bigint",
  "sum_21" : "bigint",
  "dept_name" : "varchar"
},
"outputLayout" : [ "dept_name", "sum" ],
"distribution" : "FIXED",
"outputPartitioning" : "NONE",
"partitionBy" : [ ],
"hash" : null
},
"types" : [ "varchar", "bigint" ],
"taskId" : "20150914_085716_00019_zgnuq.1.23",
"nodeInstanceId" : "05f9a498-039b-429f-a2f3-4fec49239890",
"version" : 42,
"state" : "FINISHED",
"self" : "http:// 192.168.100.12:8001/v1/task/20150914_085716_00019_zgnuq.1.23",
"lastHeartbeat" : "2015-09-14T08:57:18.371Z",
"outputBuffers" : {
  "state" : "FINISHED",
  "canAddBuffers" : false,
  "canAddPages" : false,
  "totalBufferedBytes" : 0,
  "totalBufferedPages" : 0,
  "totalQueuedPages" : 0,
  "totalPagesSent" : 0,
  "buffers" : [ {
    "bufferId" : "20150914_085716_00019_zgnuq.0.0",
    "finished" : true,
    "bufferedPages" : 0,
    "pagesSent" : 0,
    "pageBufferInfo" : {
      "partition" : 0,
      "bufferedPages" : 0,
      "queuedPages" : 0,
      "bufferedBytes" : 0,
      "pagesAdded" : 0
    }
  }
]
}
}

```


当然还可以访问 http://presto_node_ip:port/v1/task/taskId?pretty 来查看具体的 Task 详细信息，其中 `presto_node_ip` 是指运行该 Task 的 Presto 节点 IP，从中还可以查看到 Operator 的执行信息：

```
{
  "operatorId" : 1,
  "operatorType" : "TaskOutputOperator",
  "addInputCalls" : 9,
  "addInputWall" : "251.98us",
  "addInputCpu" : "0.00ns",
  "addInputUser" : "0.00ns",
  "inputDataSize" : "261B",
  "inputPositions" : 9,
  "getOutputCalls" : 0,
  "getOutputWall" : "0.00ns",
  "getOutputCpu" : "0.00ns",
  "getOutputUser" : "0.00ns",
  "outputDataSize" : "261B",
  "outputPositions" : 9,
  "blockedWall" : "0.00ns",
  "finishCalls" : 1,
  "finishWall" : "1.42us",
  "finishCpu" : "0.00ns",
  "finishUser" : "0.00ns",
  "memoryReservation" : "0B",
  "blockedReason" : null
}
```




京东大学
JD UNIVERSITY

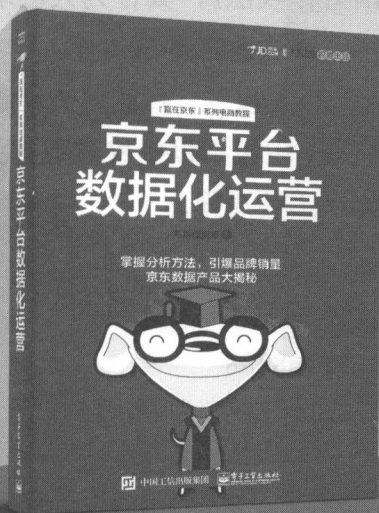
京东图书
—JD.COM—

“赢在京东” 系列电商教程

京东官方指定用书·自出版品牌图书

《京东平台数据化运营》【运营进阶版】

五大维度审读剖析 经典案例实操干货
一步解决运营难题 脱颖而出全靠它



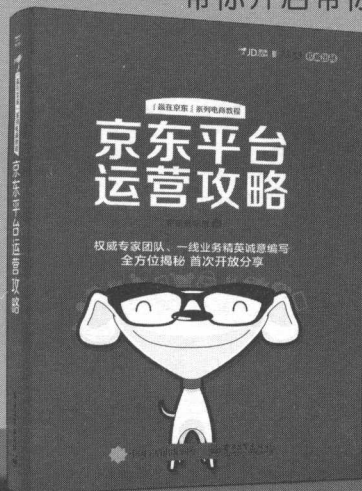
扫码立即下单

《京东平台运营攻略》【基础入门版】

平台入驻全解析 小白变身抢先机
带你开店带你飞



扫码立即下单





架构师-郭李明

2012年入职京东,一直从事与大数据分布式架构相关的技术研究,参与并主导了Presto京东开源版本的核心功能开发,致力于大数据计算的解决方案研究。



高级工程师-袁安峰

擅长Presto性能调优与Bug修复,对Presto的SQL优化器有深入研究,同时一直致力于对Presto底层实现原理的研究和探索。负责CSDN极客社区Presto板块维护,热衷于分享研究Presto技术原理的文章以及对Presto使用的心得和体会。



高级工程师-孔云龙

对HDFS、MapReduce、Hive等Hadoop生态系统中的技术有比较深入研究,在分布式查询引擎Presto和分布式消息系统Kafka的功能改造和优化方面积累了丰富的经验。目前正在京东公有云部门从事数据计算服务相关研发工作。

丛书介绍

《“赢在京东”系列技术教程》是由京东大学组织一线业务精英和权威专家团队出品的技术读物,将不断分享京东发展过程中的优势资源与先进经验,是京东唯一官方认证用书,更是品牌商家、创业者、技术从业人员等的必备参考读物。

我们将不断总结,与您共同进步。若您有意见或建议,欢迎联系我们: JD-university@jd.com

一本详细介绍Presto的书

本书由浅入深详细地介绍了Presto的安装过程、内部运行原理机制、功能特性、性能优化方法,以及在应用过程中常见的问题及解决方案等,同时详细介绍了JD-Presto版本的许多新功能及其适用的业务场景。这些功能的成功研发,使得JD-Presto版本不仅仅可用于离线大数据分析计算,还可用于多种数据源混合进行实时大数据分析计算,使一些使用其他大数据技术不能解决的业务场景有了一个全新有效的解决方案。

JD-Presto研发团队先后荣获多个奖项

在“第十届开源中国开源世界高峰论坛”上荣获“2015 COPU开源优秀项目奖”

在“第十四届中日韩三国IT局长OSS会议及东北亚开源软件推进论坛”上荣获“优秀项目技术奖”

在京东第四届CTO体系“金项奖”上荣获第一名

JD-Presto产品蕴含着团队中这群不羁的兄弟们的心血、激情和梦想,从做产品研发决定的那一刻起,就注定我们不平凡。



博文视点Broadview



京东图书
—JD.COM—



策划编辑: 张慧敏
责任编辑: 王 静
封面设计: 李 玲

出品人: 杨海峰 杨 叶
特约监制: 张 戈 邢 远 吴 迪

上架建议: 大数据/畅销书

ISBN 978-7-121-28563-9



9 787121 285639 >

定价: 69.00元